UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Agropop Web, una plataforma web para la compraventa de productos agrícolas

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Cătălin Airimițoaie

Tutor: Pedro José Valderas Aranda

Curso 2021-2022

Resum

Agropop es definex, en general, com una plataforma web de compra, venda, bescanvi, repartiment o qualsevol altre tipus de transacció entre individus o colectius, de productes agrícoles.

La plataforma es construeix sobre una sèrie de tecnologies de certa fama, destacant: *Django* com a *framework* per a desenvolupar un servidor web en el llenguatge `Python` y *Svelte* com a *framwork* pera desenvolupar un client web en el llenguatge *Typescript*.

Aquest projecte pretén documentar el procés utilitzat per aconsguir el esmentat anteriorment fins a tindre un producte mínim per a mostrar a certa cantitat d'usuaris. Per això, s'utilitzen una serie de metodologies i estrategies de treball, destacant la metodologia àgil Scrum, Kanban, Test-driven development, Trunk Based Development y altres.

També son documentats certs anàlisis tècnics del problema, tant previs, com diferents esquematitzacions abstractes del problema: casos d'ús, diagrames de clase i componentes, com posteriors: anàlisis de securitat i rendiment dels components més importants.

Paraules clau: compra, venta, agricultura, servidor web, api rest

Resumen

Agropop se define, en líneas generales, como una plataforma web de compra, venta, trueque, reparto o cualquier otro tipo de transacción entre individuos o colectivos, de productos agrícolas.

La plataforma se construye sobre una serie de tecnologías de cierta fama, destacando: *Django* como *framework* para desarrollar un servidor web en el lenguaje `Python` y *Svelte* como *framework* para desarrollar el cliente web en el lenguaje `Typescript`.

Este proyecto pretende documentar el proceso para conseguir lo anteriormente mencionado hasta el punto de tener un producto mínimo para mostrar a cierta cantidad de usuarios. Para ello, se utilizan una serie de metodologías y estrategias de trabajo, destacando la metodología ágil Scrum, Kanban, Test-driven development, Trunk Based Development y otras.

También quedan documentados ciertos análisis técnicos del problema, tanto previos, como pueden ser diferentes esquematizaciones abstractas del problema: casos de uso, diagramas de clase, diagramas de componentes, como posteriores: análisis de la seguridad y rendimiento de los componentes más importantes.

Palabras clave: compra, venta, agricultura, servidor web, api rest

Abstract

Agropop is defined, in general terms, as a web platform which provides its users with the ability to sell, buy, barter, distribute or any other type of transaction between invidivuduals of produce-based products.

The platform is built upon a series of famous technologies, highlighting: *Django* as a framework to develop the web server using the `Python` language, and *Svelte* as a framework to develop the web client using the `Typescript` language users. To do so, a series of metodologies and work strategies are being used, highlighting the agile methodology Scrum, Kanban, Test-driven Development, Trunk Based Development and others.

Other technical analysis will also be provided. These will be either previous to the development process, like building different abstract schemas and diagrams to predefine behaviour, like use case, class and componentes diagrams, or posterior: security and performance analyses of the most important componentes.

Key words: sell, buy, agriculture, web server, rest api

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

La agricultura, aún siendo una de las bases de la sociedad humana, ha ido evolucionando, como industria, hacia unas condiciones insostenibles para los trabajadores de la tierra.

"[...] Un primer periodo, desde la segunda mitad de los años sesenta a la primera de los años noventa, donde las gananacias de los granjeros y trabajadores de la tierra era claramente insuficiente para cubrir el gasto familiar, en términos medios rurales o nacionales. Esto explica por qué las diferencias entre estándares de vida entre las esferas rurales y urbanas de la época eran tan significativas. Un segundo periodo, que empezó en la segunda mitad de los años noventa, donde los ingresos de los trabajadores sí cubrían los gastos familiares, pero solo gracias a la destrucción de empleo y el cese de actividad de varias granjas, tal como veremos más abajo. Los ingresos de la agricultura no han mejorado, de hecho continuaron en su deterioro, estos simplemente fueron distribuidos entre menos granjeros y trabajadores de la tierra. "(Manuel González de Molina, 2019)

Este hecho tiene muchas causas. Una de ellas es que los agricultores llevan años dejando de tener control sobre los precios de sus propios productos ya que estos vienen definidos por una larga lista de intermediarios que no realizan ningún proceso transformatorio sobre el producto para añadir valor, más allá de distribuirlo globalmente, con todo lo que ello implica:

"Las industrias que proveen, las sociedades comerciales que dan acceso al mercado y los bancos que financian inversiones capitales tienen decisiva influencia sobre los agricultores. Es el producto de políticas de los gobiernos domésticos y leyes y acuerdos de comercio internacional, así como los desarrollos en ciencia y tecnología y el esfuerzo local sobre régimenes de trabajo y tierra. "(Isett, Miller, 2016)

1.1 Motivación

Sirva este apartado como descripción de objetivos sociales sobre los que Agropop basa su existencia, pues esta viene motivada por las conclusiones de la introducción anterior.

Agropop pretende dar una alternativa al método clásico de venta de productos agrícolas donde los trabajadores de la tierra se ven forzados a renunciar a gran parte de sus beneficios a terceros, presionados por intermediarios y otros agentes mercantiles. Asimismo, también ofrece una alternativa completamente no-capitalista a este problema, dando la posibilidad de utilizar la plataforma también para el **reparto** libre o trueque, pudiendo eliminar la motivación monetaria. Esto es útil para organizaciones sin ánimo de lucro, pudiendo dar más visibilidad a proyectos de huertos urbanos y similares.

Todo esto se consigue implementando un servicio que presta conceptos de otras plataformas sociales como *Mastodon* (federación) y *Wallapop* (intercambio de bienes).

1.2 Objetivos

El principal objetivo de Agropop es proveer una plataforma de código libre[1] que cualquier organización pueda implementar para poder ofrecer servicios de compraventa, reparto o trueque de productos agrícolas.

Desde el punto de vista de un usuario cualquiera, este objetivo se puede desglosar en los siguientes ítems:

- Poder crear y borrar un usuario
- Poder crear una tienda compuesta por varios kioskos o unidades organizativas de diferentes productos. Por ejemplo, una tienda de un usuario que venda solo frutas puede organizar la misma en kioskos según cada temporada
- Poder añadir y eliminar productos a esos kioskos
- Poder utilizar un chat para establecer comunicaciones con otros usuarios. Estos chats están ligados están ligados a un producto en concreto
- A nivel de vendedor:
  - Dar la posibilidad de categorizar los productos en kioskos ligados a una tienda única que pertenece al vendedor, teniendo, por ejemplo, una tienda de venta de frutas pero diferentes kioskos para cada temporada
  - Poder establecer los kioskos en un mapa interactivo para que los compradores puedan ubicarlos
- A nivel de comprador, además de lo que lo anterior implica, también se debe poder tener acceso a un mapa general donde se muestren todas las tiendas y kioskos disponibles, filtrados según diferentes parámetros

Desde el punto más técnico, el proyecto cumple los siguientes objetivos:

- Ofrecer una plataforma de código abierto disponible para cualquier organización interesada
- Cumplir con estándares a la hora de la implementación. Aunque serán explicados en detalle más tarde, requieren una mención aquí el estándar definición de Application Programming Interface (API) web utilizado: *OpenAPI 3.0*
- Ofrecer una decente calidad de código mediante técnicas modernas de integración y despliegue continuos, lo que implica implementar la pirámide de pruebas[2]

[1] https://www.gnu.org/philosophy/free-sw.en.html

[2] La pirámide de pruebas es una agrupación donde estas están organizadas por su granularidad: en la base están los tests unitarios, seguidos por los tests de integración, que comunican con las plataformas externas, y los tests punto a punto, que prueban todos los flujos de principio a fin incluyendo la interfaz de usuario

1.3 Metodología

La metodología de trabajo utilizada es Scrum basada en principios ágiles. El cuarto principio básico del desarrollo ágil es el de la respuesta a los cambios por encima del seguimiento de un plan riguroso. Este principio es y ha sido fundamental para el desarrollo de Agropop, ya que al inicio del proyecto existía mucha incertidumbre tecnológica,lo que ha ayudado a poder planificar el trabajo teniendo en mente esta incertidumbre.

En cuanto al aspecto más técnico, la metodología de programación usada se basa en TBD[3] por una razón muy parecida a la anterior.

Ambas metodologías se verán explicadas en más detalle en el apartado Plan de trabajo.

1.4 Estructura de la memoria

Este trabajo incluye, principalmente, documentación sobre los procesos abstractos y técnicos para el desarrollo de un servicio mínimo de comercio electrónico entre particulares.

Para ello se pone a disposición, en un primer momento, un análisis del estado actual de mercado para ver qué soluciones existentes resuelven este problema, aunque sea tangencialmente. A continuación se realiza una retrospectiva sobre el plan de trabajo realizado, explicando en cierto detalle las metodologías utilizadas, un análisis material y económico del proyecto, así como un pequeño estudio de la seguridad virtual que el problema puede plantear.

A continuación se realiza una descripción en cinco capítulos exclusivamente del problema, abordando el diseño, el desarrollo, la implantación, las pruebas y desarrollos futuros.

Mencionar que este trabajo hace gran uso de tecnologías de código libre desarrolladas por las comunidades creadas alrededor de las tecnologías base empleadas. Este tipo de paquetes o librerías se ven mencionados con asiduidad a lo largo del trabajo pero la primera vez que ocurre este hecho, suele existir o un enlace a la página principal, o una explicación breve, ambas opciones o una referencia en el glosario de definiciones final.

1.5 Análisis ético y legal

1.5.1. Tratamiento de datos

Existen ciertos datos de carácter sensible que Agropop colecciona y modifica. En líneas generales, estos datos son los necesarios para gestionar usuarios y sus operaciones comerciales, destacando nombre, email y datos bancarios. Sin embargo, Agropop no es una platafora que exista por sí misma, necesita ser implantada por terceros, lo que implica que la responsabilidad de estos datos cae sobre dichos implantadores, que, generalmente, y si se encuentra en Europa, deberían adherirse al General Data Protection Regulation (GDPR)[4], así como mostrar al usuario que se van a guardar y tratar cookies de sesión y otros. Por otro lado, en cuanto a los datos bancarios, Agropop pretende dejar esta responsabilidad a terceros como Stripe o PayPal.

[3] https://trunkbaseddevelopment.com
[4] https://gdpr-info.eu/

1.5.2. Propiedad intelectual y software libre

La propiedad intelectual de Agropop les pertenece a sus desarrolladores pero el proyecto está desarrollado bajo la licencia GNU Public License (GPL)v3[5] lo que implica que todos los usuarios tienen ciertos derechos y deberes con lo que respecta al código fuente del mismo: a grandes rasgos, acceso completo al mismo así como la posibilidad de modificación y redistribución, siempre que la modificación no implique la modificación de la licencia o uso de tecnologías propietarias, además de la obligación de aplicar la misma licencia para cualquier cambio realizado por terceras partes y todo lo que ello implica.

[5] https://www.gnu.org/licenses/gpl-3.0.en.html

CAPÍTULO 2
Estado del arte

En este apartado abordaremos tecnologías y plataformas actuales que ya cumplen todos o algunos de los objetivos funcionales mencionados anteriormente y especifcaremos qué valor añade Agropop a estas soluciones.

En contrapartida a las plataformas tradicionales, Agropop ofrece al vendedor la posibilidad de poder organizar sus productos en kioskos y una mejor integración con la vista de mapa.

Por otro lado, como Agropop es una plataforma de código libre, y nuevamente comparando con Wallapop, cada organización tiene que y puede implementarla según sus propias consideraciones eliminando o añadiendo funcionalidades. Este último punto también implica que toda la responsabilidad del albergue de la plataforma cae sobre la organización implementante y no sobre los desarrolladores de Agropop, a priori.

CAPÍTULO 3
Planificación

3.1 Plan de trabajo

3.1.1. Agilidad vía Scrum

Como se ha explicado con anterioridad, la metodología de trabajo ha sido Scrum con puntos. Esto permite tener un modelo de trabajo ágil y con tolerancia a horarios de trabajo flexible, aunque en líneas generales se ha logrado establecer un número de puntos común a casi todos los sprints como fuerza de trabajo disponible: 90.

Cómo funciona Scrum

A grandes rasgos, Scrum funciona definiendo, entro otros, proyectos, sprints y un backlog, donde un proyecto encapsula un número de sprints para conseguir un objetivo en un periodo de tiempo. Por ejemplo, para este trabajo se ha definido el proyecto Minimum viable product (MVP), donde se busca desarrollar el producto mínimo mostrable al usuario en la menor cantidad de tiempo. Cada proyecto define una serie de sprints con los cuales plantea el desarrollo ininterrumpido del proyecto. Un sprint es básicamente una unidad de tiempo en la cual gestionar ciertas tareas o *historias de usuario*.

Una historia de usuario es la definición del pedazo mínimo de comportamiento de un sistema informático. Existe una larga lista de definiciones de qué es y qué es una buena una historia de usuario. Una de ellas es Independent, Negotiable, Valuable, Estimable, Small, Testable (INVEST), donde:

- *Independent* hace referencia a que las historias de usuario, idealmente, deberían poder ser entregadas en cualquier orden
- *Negotiable* significa que los detalles de la historia son definidos por la mayoría de implicados, desde programadores a clientes
- *Valuable*: la funcionalidad tiene que aportar valor a los clientes o usuarios
- *Estimable*: los programadores deben poder hacer una estimación razonable sobre el ciclo de vida de la misma
- *Small* significa que estas deberían ser mínimas en el tiempo, generalmente a unos cuantos días por persona
- *Testable* hace referencia al hecho de que los programadores puedan verificar la funcionalidad

Un sprint es organizado cogiendo este tipo de unidades de descripción de comportamiento del backlog en un orden establecido para cumplir con el proyecto. A cada historia de usuario se le asigna un punto de una serie que en el caso de Agropop, sigue la secuencia de Fibonacci[1] mínimamente modificada. Los puntos encapsulan **dificultad**: esto es, una combinación de todos los factores que pueden hacer que la tarea sea más difícil de desarrollar. La mayoría de los factores que cumplen esto son los contrarios a los factores que definen INVEST. Teniendo esto en cuenta, qué valor de dificultad signifca, por ejemplo, el número 13, depende únicamente de la experiencia del equipo según avanza el desarrollo.

Esta organización funciona siempre y cuando el equipo tenga continua comunicación con el cliente, para poder, además de establecer el órden e importancia de tareas y sprints, gestionar qué hacer cuando las cosas *van mal*.

Como cada tarea tiene asignado un punto, en un sprint dado, la suma de estos puntos define los puntos del sprint. Esta cantidad se ha ido descubriendo o calculando a lo largo de los dos primeros sprints pero también ha ido lo que significa el punto mínimo de dificultad a medida que el equipo se ha familiarizado con las tecnologías utilizadas y los problemas a resolver.

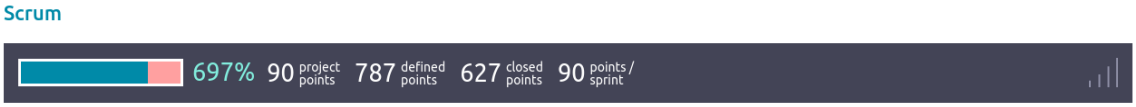

Figura 3.1: Vista general del proyecto en el Sprint 7

Podemos observar en Figura 3.1, una captura de la herramienta utilizada para llevar seguimiento del proyecto, cómo se ha calculado la cifra mencionada anteriormente. Podemos observar cómo el proyecto, en el Sprint 7, está un 627/787 terminado para realizar un MVP. Según este número, dicho MVP, en este punto, está en mayor o menor medida al 80 % finalización. El porcentaje de más a la izquierda en la figura es debido a que la herramienta añade como superávit los puntos de cualquier tarea añadida al proyecto después del primer sprint, y es tan alta porque aunque se ha creado un backlog inicial, la planificación de los sprints se ha realizado con dos de ellos en mente según el proyecto evolucionaba, lo que ha conllevado a mucho descarte y creación de nuevas tareas en el backlog.

En un aspecto menos abstracto, se ha definido un sprint, como ya hemos dicho antes, a 90 puntos de trabajo realizados en un periodo de tres semanas, teniendo el MVP en seis meses (ocho sprints a tres semanas, asumiendo que un mes tiene cuatro semanas).

Los primeros sprints

El primer sprint, número cero, no viene contemplado en el anterior cálculo porque es un sprint de definición abstracta del proyecto, donde se desarrollaron: el modelo de dominio, los diagramas de casos de uso y clases básicos así como la creación de los repositorios y elección de tecnologías. Este sprint duró una semana.

Aunque el sprint cero es el primero, a continuación no lo tendremos en cuenta a la hora de nombrar los sprints para evitar confusiones con los números de los mismos, siendo, de manera efectiva, a partir de aquí, el Sprint 1 el primero.

Asimismo, al primer sprint, encargado de plantar las bases del frontend, se le asignaron 88 puntos y se lograron cerrar todos pero finalizó abruptamente unos días antes

[1] 0, 0.5, 1, 2, 3, 5, 8, 10, 13, 20, 40

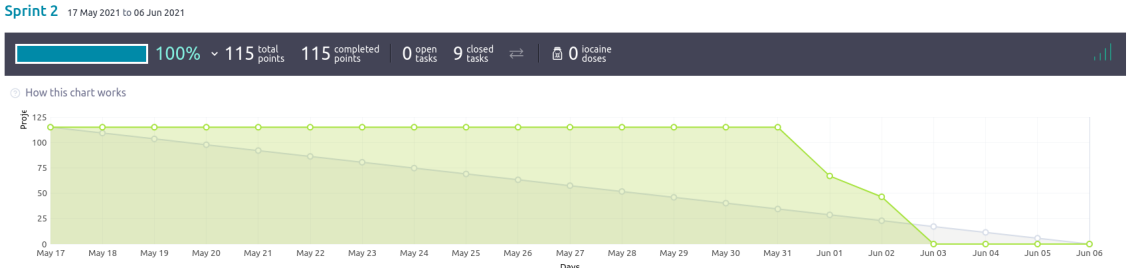

Figura 3.2: Vista final del sprint 2

Figura 3.3: Vista final del sprint 2 - desglose

de lo debido, lo que provocó que el segundo sprint, encargado de esbozar las bases del *frontend*, fuese planeado con 115 puntos. Sin embargo, como esta tecnología era completamente nueva para el equipo, se sobreestimaron la mayoría de tareas. Tal como se observa en Figura 3.2, costó bastante empezar a cerrar tareas por lo novedoso de las tecnologías pero en cuanto esto fue superado, casi todas fueron cerradas de golpe. Este hecho también indica que las historias de usuario no fueron lo suficentemente desglosadas en subtareas.

Los sprints restantes

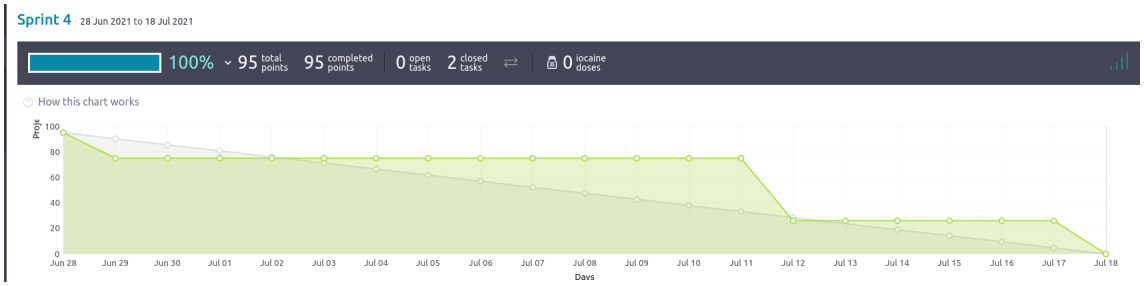

Figura 3.4: Vista final del sprint 4

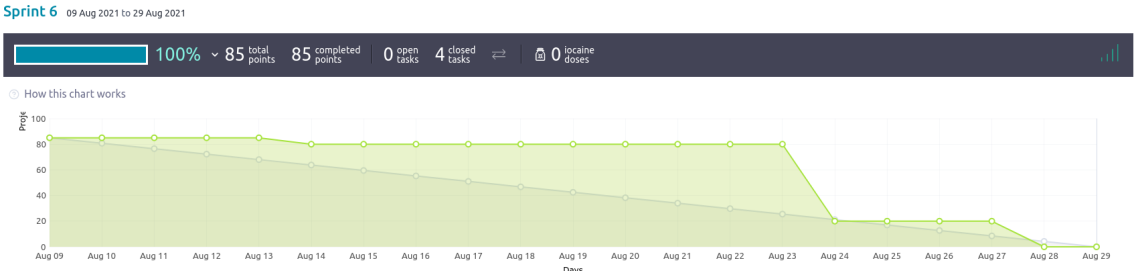

Figura 3.5: Vista final del sprint 6

Podemos observar según Figura 3.4 y Figura 3.5 como la evolución de los mismos es más natural, aunque se sigue pudiendo observar un problema de desglose de historias de usuario algo pobre.

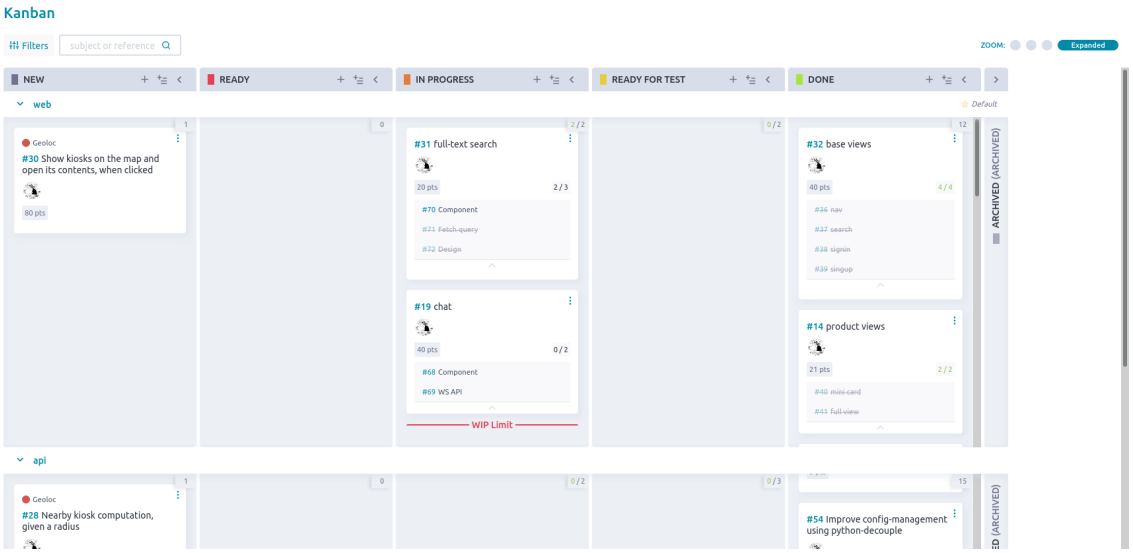

Figura 3.6: Vista de Kanban en el sprint 7

3.1.2. Microgestión vía Kanban

Se ha utilizado Kanban para (micro)gestionar las historias de usuario de cada sprint, habiendo un tablero para cada uno. Esta metodología permite utilizar dos herramientas para hacer más fácil todo este seguimiento: *swimlanes* y el número Work In Progress (WIP).

Las *swimlanes* permiten separar el tablero en dos, en nuestro caso, frontend y backend, y este hecho es importante cuando se combina con la segunda herramienta, el número WIP, el cual permite definir un número máximo consecutivo de tareas para una columna del tablero determinada. Esto a nivel práctico implica que un equipo tiene un máximo número de tareas a las que puede dedicar un estado, evitando así cuellos de botella cuando el equipo suele favorecer solo una de las columnas. Como a veces era necesario trabajar en ambos frentes a la vez, un WIP sin *swimlanes* tendría que ser innecesariamente grande -lo que podría provocar problemas cuando no se trabajase en ambos frentes- para poder albergar este número de tareas, pero con la combinación de las dos, se puede tener uno para cada *swimlane*

En la Figura 3.6 se puede observar cómo para el *swimlane* del *frontend*, llamado web, se ha alcanzado el WIP máximo para la columna de *En progreso*, el cual es de dos.

Si hubiera una tarea en progreso en la parte *backend* esto no afectaría al *WIP* web.

3.1.3. TDD y TBD

Una ventaja de utilizar una metodología ágil es que se adopta más fácilmente que otras a lo que se conoce como *Test-driven development*, un proceso de desarrollo donde los requerimientos definen pruebas de aceptación y la programación de la lógica se realiza con dichas pruebas en mente.

La tecnología de sincronización del código a utilizar es *Git*[2], lo que supone una serie obstaculos a la hora de definir un sistema de trabajo colaborativo sobre dicha plataforma. El más famoso suele ser *GitFlow*.

[2] https://git-scm.com/

GitFlow define una serie de ramas no efimeras que actúan como conducto del ciclo de madurez de un producto: la rama principal tiene el código más maduro y probado, mientras que la rama de desarrollo tiene el código menos maduro. Entre ambas pueden haber cualquier número de ramas no efímeras según, por ejemplo, el número de niveles de testeo por el que el código pueda pasar. Además de esto, el propio desarrollo se realiza sobre lo que se conoce como ramas de nuevas características *-feature branches-*, ramas efímeras que se fusionan con la rama no efímera menos madura, la rama de desarrollo, y a partir de ahí el código continua subiendo hacia la ramas más maduras según pasa por el proceso de aceptación.

Pero este proceso es altamente tedioso, sobre todo si se desea desplegar código rápida y asiduamente: la rama más madura puede estar varias semanas sin ser actualizada porque una rama anterior puede tener un cuello de botella en una de las pruebas de aceptación, sobre todo si estas son manuales. Además, también puede añadir mucha confusión porque en cualquier momento dado pueden existir un gran número de ramas o que muchas de ellas estén desactualizadas y ya no se usen.

Para combatir esto, Agropop usa otro proceso llamado *Trunk Based Development*, el cual solo puede existir en tándem con un proceso de testeo robusto en paralelo, por lo que la mezcla con TDD es perfecta. Esto es porque TDD solo define una rama no efímera, la llamada `trunk` o rama principal. e invita a resistir cualquier presión por crear ramas donde su vida sea larga: en TBD, una rama debería servir un desarrollo mínimo de máximo uno o dos días de trabajo. Al cabo de esto, debería ser fusionada con la rama principal y ser borrada. Pero este proceso solo es posible si:

- las pruebas son completamente automáticas
- hay un nivel de cobertura considerable
- existe una plataforma de desarrollo continuo e integrado que pueda garantizar lo anterior
- el código de la rama principal no es desplegado al entorno de producción al que el cliente tiene acceso
- cuando la rama principal llega a un cierto punto de madurez, se crea, a partir de la misma, una entrega *-release-* y es ese el código que se expone al público

3.2 Presupuesto

El trabajo hecho no es uno enfocado necesariamente a la monetización por el tipo de licencia aplicado. Aún así podemos calcular qué presupuesto es necesario para realizar el MVP. Según datos de *LinkedIn*, el salario medio de un programador en el estado español es de 22.000€ para un horario de 40h semanal. Teniendo en cuenta que este proyecto ha tenido una duración de seis meses a 25h por semana, el salario de un solo programador medio sería de 6800€. Sumando un ordenador mínimo para el desarrollo web de 500€, daría un presupuesto de 7300€.

Un tema más interesante es el modelo de negocio asociado a proyectos de código libre, siendo este uno basado en el mantenimiento, despliegue y extensión de la aplicación base, para terceros. Esto viene en contraposición a basar el negocio en licencias o ventas a consumidor directo.

Este modelo puede tener muchos enfoques, siendo uno de ellos el basado en proyectos -y así adoptar la metodología Scrum al propio negocio- de duración indefinida

según un número de sprints pactados con el cliente, siempre ampliable. Se podría, así, vender sprints de horas variables según las necesidades del cliente, teniendo un modelo de negocio flexible para cubrir varios escalones de mercado. Una escala interesante sería, estableciendo una base de 65€ por hora de trabajador.

- Sprint de 40h por semana con un trabajador y una duración de tres semanas: 7800€/sprint
- Sprint de 25h por semana con un trabajador y una duración de tres semanas: 4875€/sprint
- Sprint de 15h por semana con un trabajador y una duración de dos semanas, aumentando el precio base a 75€/h: 3375€/sprint

Con lo que respecta a la propia infraestructura, este aspecto puede ser flexible: la empresa cliente puede proporcionar su propio hardware o albergue *cloud*, o Agropop puede facilitar infraestructura con un coste adicional. Qué precio tiene la infraestructura depende de las necesidades y expectativas del cliente.

Concluyendo, si se logra encontrar zonas geográficas donde Agropop resulta interesantes, se puede ofrecer servicios de empresa a empresas para implantar la plataforma, teniendo un modelo de negocio rentable.

CAPÍTULO 4
Análisis del problema

Este apartado mostrará diferentes análisis funcionales del proyecto para poder establecer métricas de seguridad, marcos legales y otros, así como establecer las técnicas, diagramas y modelado de los distintos niveles de conocimiento con el que se puede analizar el problema.

Como ya se ha mencionado, Agropop es desarrollada utilizando una metodología ágil, lo que implica que el análisis de requisitos no es un objeto estático. Este evoluciona según nuevas necesidades y características que se han visto implicadas en el desarrollo que, bien por necesidad o bien porque se ha apreciado un cambio que mejora el estado la mayoría de diagramas y modelos se van a mostrar varias versiones para observar cómo actual en un momento dado, han hecho que las bases abstractas del mismo también cambien.

Estas evoluciones están ligadas al concepto de sprint, explicado en el apartado de Plan de trabajo. Esto no implica que haya una evolución de cada análisis por cada sprint, aunque puede darse el caso.

4.0.1. Casos de uso

Los diagramas de caso de uso muestran los requerimientos del problema como acciones llevadas por actores.

Al igual que en el apartado del proyecto, los casos de uso iniciales son triviales y sirven para definir lo necesario a implementar en los dos o tres primeros sprints.

Podemos observar en la Figura 4.1, cómo un usuario sin cuenta puede crear una para convertirse en usuario autenticado, pudiendo luego realizar operaciones Create, read, update, delete (CRUD) sobre diferentes ítems: productos, kioskos y su propio perfil. Además, existen dos actores más: el kiosko, el cual es encargado de vender un producto, y el administrador, el cual puede realizar operaciones CRUD sobre cualquier ítem.

A medida que el desarrollo avanza, el diagrama de casos de uso adquiere una complejidad exponencial por lo que sufre un número de cambios significativos por el camino, adquiere un estado casi definitivo pues incluye casos de uso que abarcan un alcance mayor al de esta memoría. Se verán explicados en el Sección 8.2.

La Figura 4.2 muestra el subsistema de interacción entre comprador y vendedor. En primer lugar, se puede observar cómo ahora existe un mayor número de actores y una subclasificación (generalización) de los mismos: el usuario autenticado puede actuar como comprador y vendedor y sus acciones son diferentes. El actor comprador puede realizar operaciones CRUD sobre sus ítems (productos, kioskos y tienda) mientras que el usuario comprador puede ver los ítems de otros usuarios. Para iniciar una compra, un

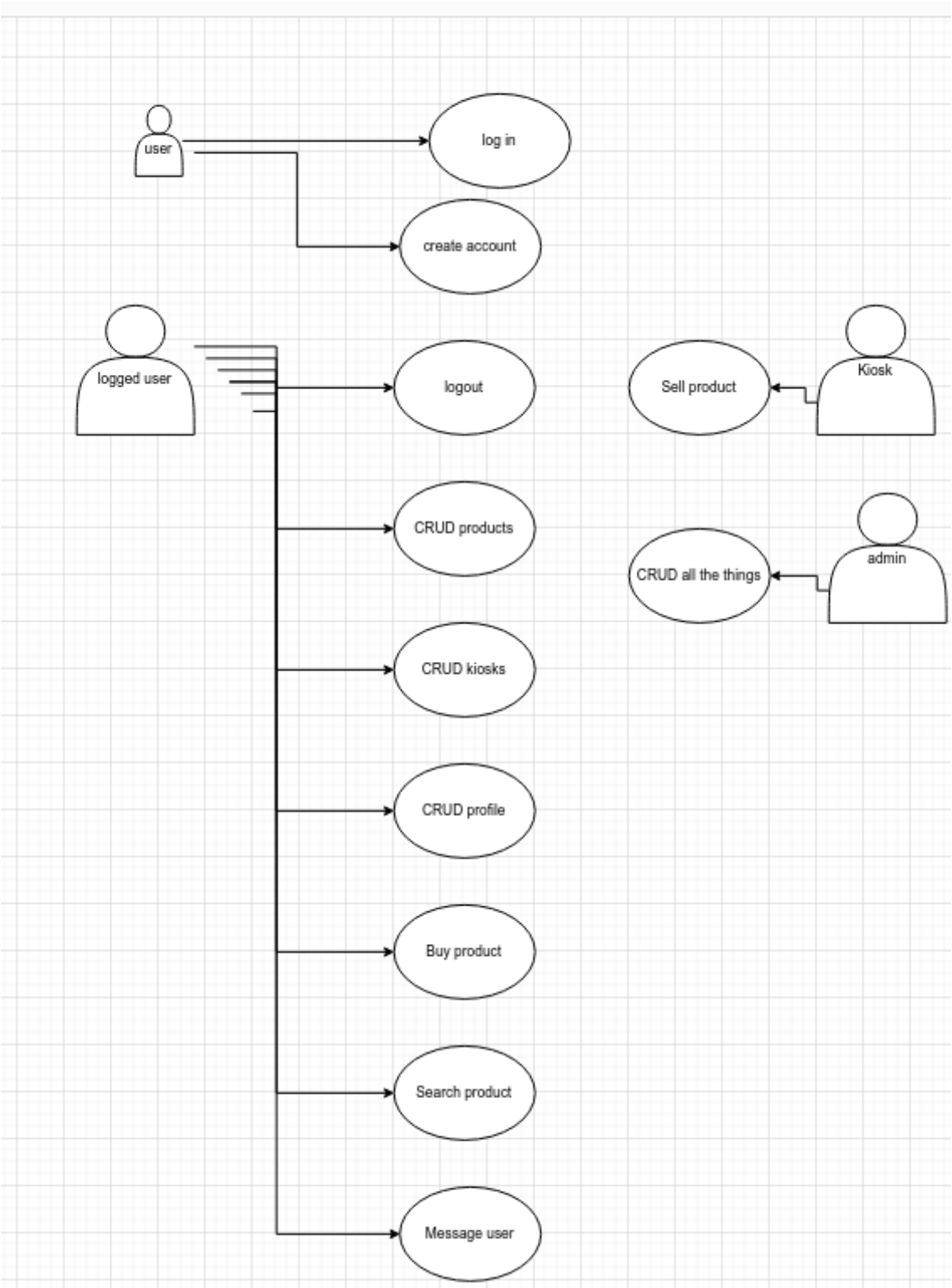

Figura 4.1: Diagrama de casos de uso del sprint 0

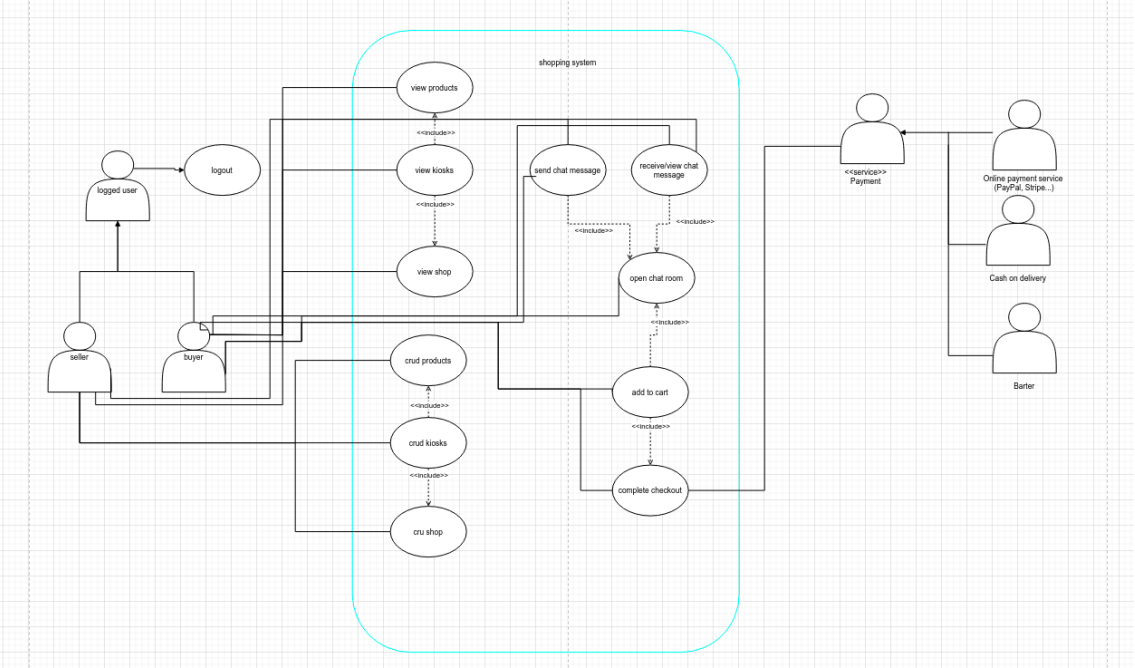

Figura 4.2: Diagrama de casos de uso del sprint 5

usuario comprador debe abrir una sala de chat y a partir de ahí el proceso de compra-venta continua.

La Figura 4.3 muestra el sistema de búsqueda de ítems, el cual incluye también las vistas de mapa. Podemos ver cómo existe una búsqueda global en ambos casos, tanto en la búsqueda normal cómo en el filtrado de mapas, aunque el segundo no incluye búsqueda por productos. La búsqueda normal global es una búsqueda *full-text*[1]

[1]En este caso *Full-text search* hace referencia a una técnica de búsqueda que permite filtrar y recuperar datos de una base de datos independientemente de cómo los datos de esta estén clasificados o tabulados, o lo que es lo mismo, la búsqueda no es filtrada a priori según metadatos. En nuestro caso, esto permite buscar un término en todas las tablas de la base de datos a la vez, independientemente de si el término es propio de un producto, kiosko o tienda

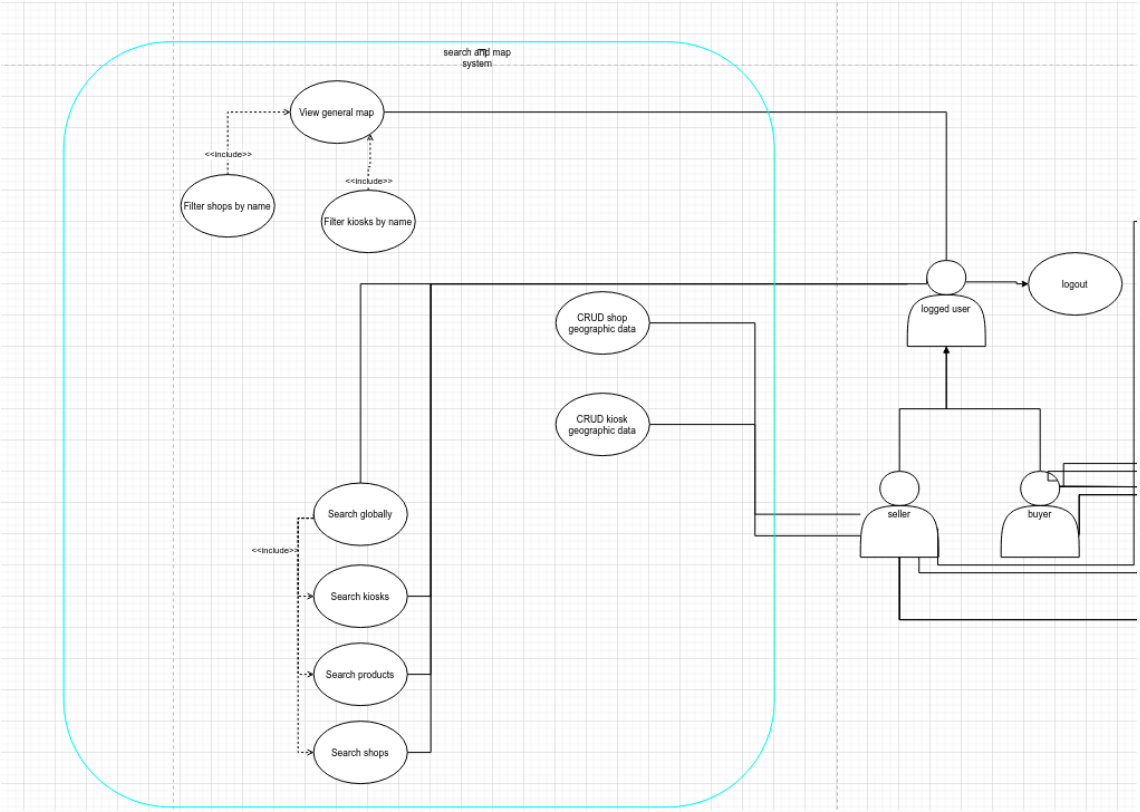

Figura 4.3: Diagrama de casos de uso del sprint 5-2

4.0.2. Diagrama de clases

El modelo de datos viene dictado por el Object-relational mapping (ORM) por lo que se podrá observar que coincide mucho con la arquitectura del backend, por lo que resulta interesante mencionar que solo se mostrarán los apartados relevantes, o lo que es lo mismo, en este apartado o en los que traten la arquitectura del backend no se tratarán lo obligado por *Django* como vistas o *parsers*.

Al igual que en apartados anteriores, el diagrama Unified Modeling language (UML) ha ido evolucionando según el desarrollo ha avanzado.

Se puede observar cómo en un primer momento y según la Figura 4.4, se define una clase base, `BaseModel`, la cual define los atributos más básicos para los demás modelos[2]. No aparece en este diagrama por lo mencionado anteriormente, pero la clase `BaseModel` hereda a su vez de la clase base modelo de *Django*. Podemos observar, luego, las relaciones entre cada ítem: dado un usuario[3], este tiene una tienda asignada y esta, a su vez, un número n (cero, uno o muchos) de kioskos, relación que comparte un kiosko cualquiera con sus productos. La clase `Media` hace referencia a una encapsulación de una ruta a un archivo y su tipo Multipurpose Internet Mail Extensions (MIME), un estándar de metadato para establecer el tipo que tiene un fichero. Por ejemplo, `image/png`, lo cual es útil para guardar archivos multimedia relacionados a diferentes modelos.

En el apartado futuro, Manejo de las URL, se habla de cómo gestionar la nomeclatura de los recursos en un espacio de URL por lo que el modelo base ha actualizado para añadir un nuevo atributo tipo *handler*, el cual es utilizado para identificar un recurso. El *handler* tiene que ser único y puede ser autogenerado por el sistema. Además de esto,

[2] *Django* utiliza el término "modelo"para referirse a lo que comúnmente entenderíamos como clases

[3] existen dos clases de usuario por limitaciones de *Django*: crear otra clase y relacionarla uno a uno con la clase `User` básica de *Django* es la manera más recomendable de extender dicha clase

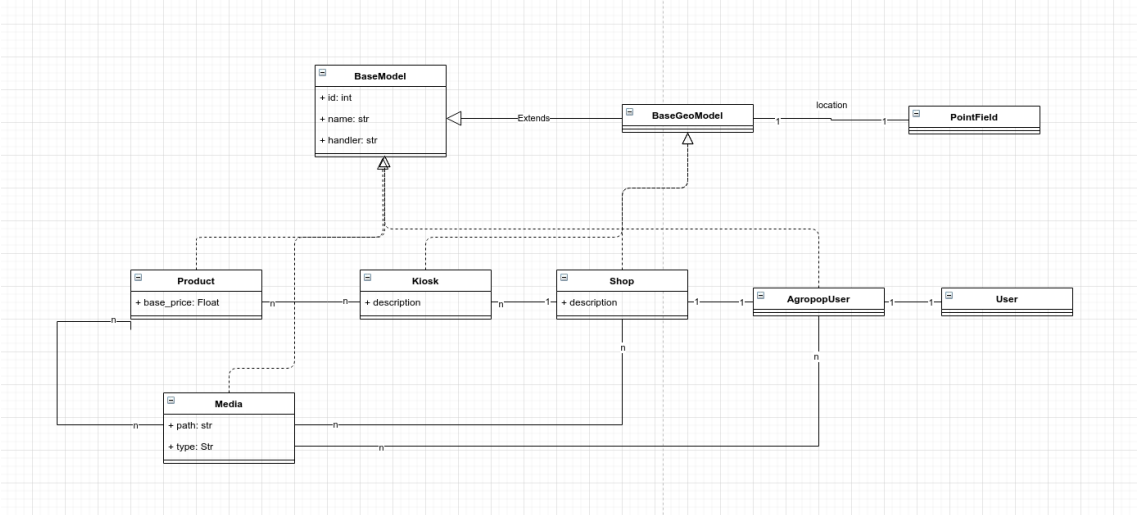

Figura 4.4: Diagrama UML en el sprint 2

se ha añadido un nuevo modelo base llamado `BaseGeoModel` para que los modelos que necesiten geolocalización tengan esta característica directamente; más específicamente, el modelo base de geolocalización implementa un atributo `location` del tipo `PointField`, el cual es una abstracción Geographic Information System (GIS) de un punto geográfico.

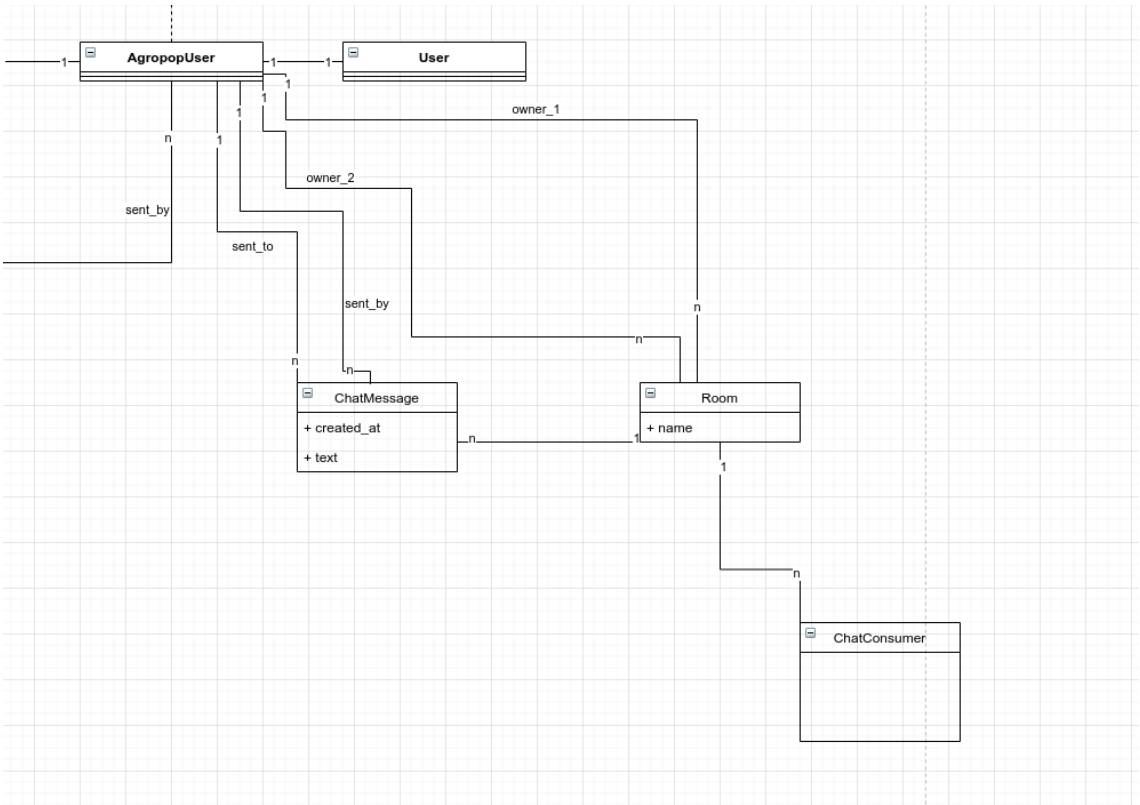

Figura 4.5: Diagrama UML en el sprint 3

Una vez se implementaron las clases anteriores, se añadió al esquema las clases que componen el chat. En la Figura 4.5 se puede ver cómo las clases necesarias para implementar el chat interactuan con el usuario. Aunque más adelante se explicará en más detalle, sobre todo la parte de la implementación, se puede observar como existe una sala que contiene los mensajes, y ambos pertenecen a dos usuarios, puesto que dos usua-

rios interactúan con dichos modelos. La clase `ChatConsumer` hace referencia a la clase que implementa la lógica.

CAPÍTULO 5
Diseño de la solución

En este capítulo vamos a exponer cómo se ha esbozado el proyecto a nivel abstracto. Como se verá en un apartado siguiente, Identificación de soluciones posibles, el proyecto está separado en dos partes muy diferenciadas, frontend y backend, por lo que estas dos partes van a suponer también la diferenciación principal de este capítulo, aunque acaben compartiendo subapartados. Sin embargo, la parte fundamental de este capítulo es el modelo de datos, el cual se define a nivel de backend y se utiliza en el frontend.

5.1 Modelo de datos

El modelo de datos muestra la disposición final de los datos en la base de datos: estructura, atributos y relaciones. En la Figura 5.1 podemos ver una vista completa de todas las tablas, con sus atributos y relaciones, del proyecto. Sin embargo, esta figura es ininteligible por cuestiones del formato usado en esta memoria, así que a continuación haremos énfasis en los dos subdominios más importantes: la tienda y el chat.

En la Figura 5.2 observamos la relación entre las tablas que componen el sistema de tiendas, las cuales, por cómo funciona *Django*, vienen prefijadas por el término `shop_`, luego, por ejemplo, la tabla de los kioskos se llama `shop_kiosk` y tiene un atributo `shop_id` como clave ajena a la tabla de tiendas, la cual es denominada `shop_shop`. Como existe una relación de muchos a muchos entre las tablas de producto `-shop_product-` y kiosko `-shop_kiosk-`, se ha creado una tabla intermedia `shop_kiosk_products` que relaciona los dos.

Por otro lado, la Figura 5.3 muestra las relaciones entre las tablas que componen el sistema de chat, donde vemos, por ejemplo, como la tabla de mensajes `-chat_chatmessage-` se relaciona con una sala `-chat_room-` y dos usuarios `-shop_agropopuser-`, remitente y emisor, mediante los atributos `room_id`, `sent_by_id` y `sent_to_id`.

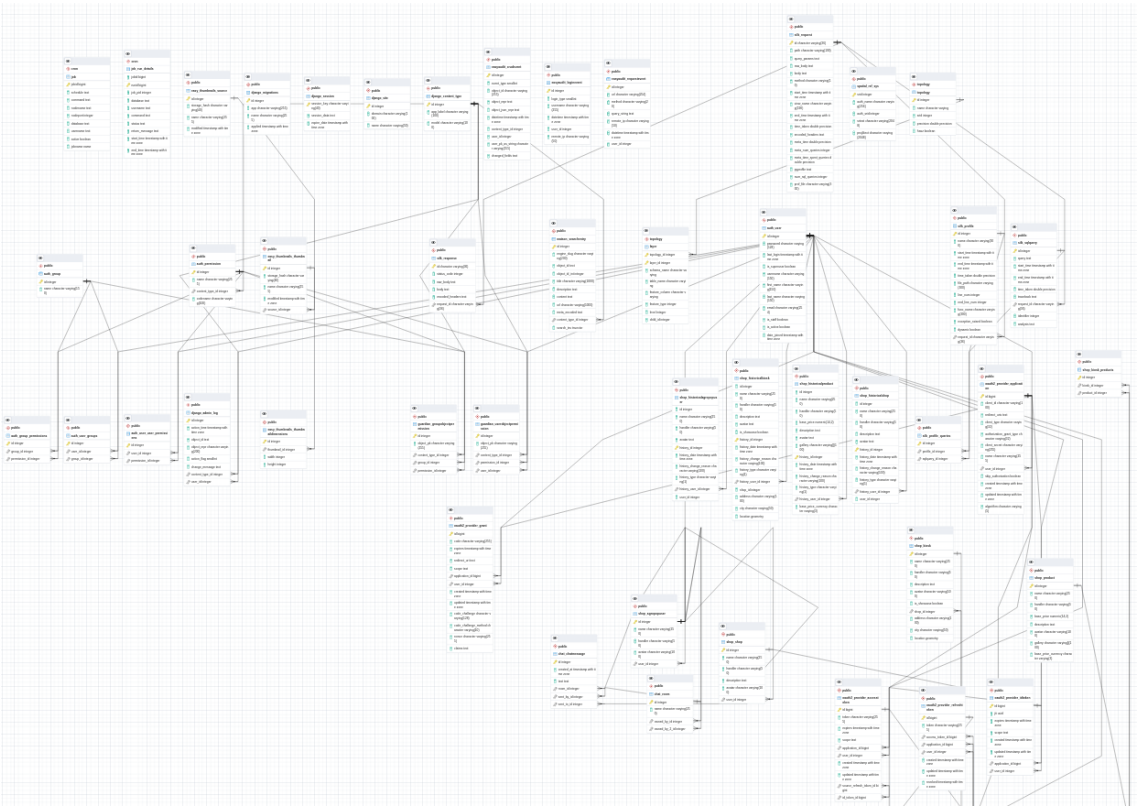

Figura 5.1: Modelo de datos completo

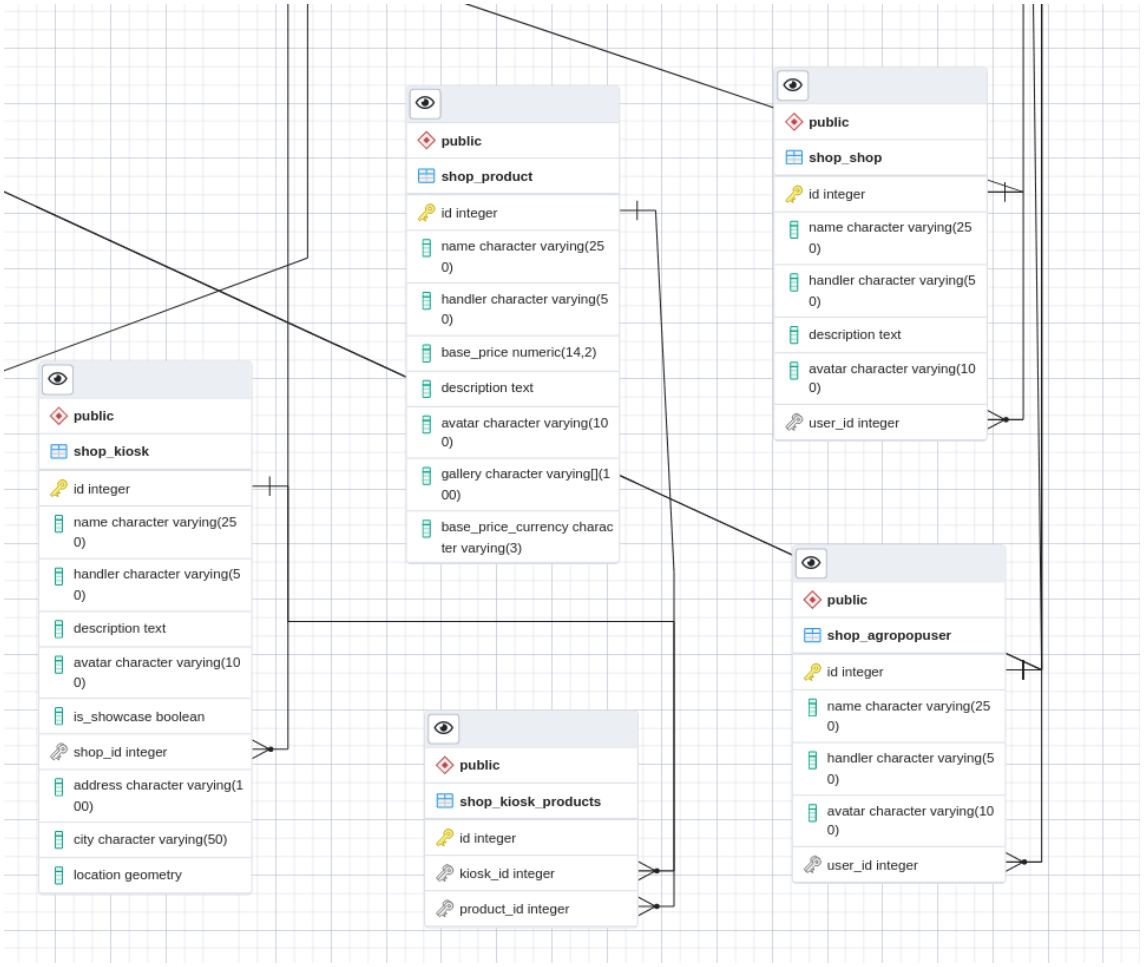

Figura 5.2: Modelo de datos - dominio de la tienda

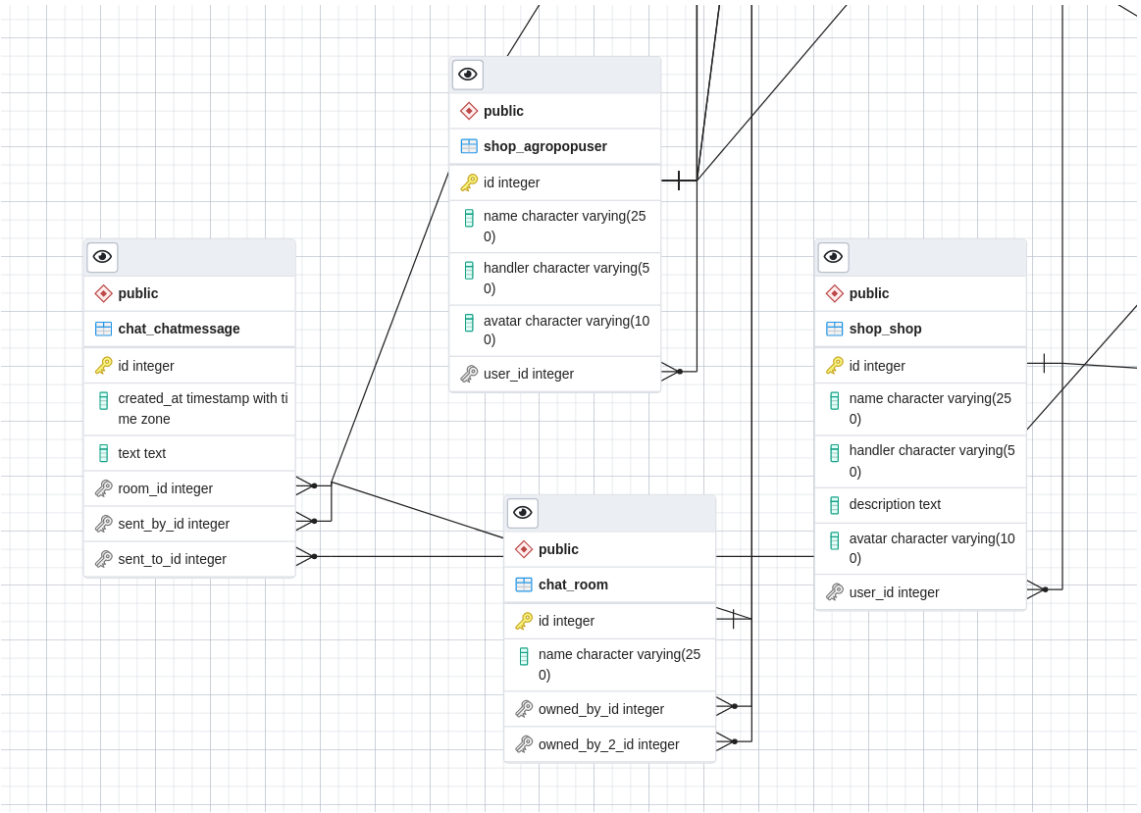

Figura 5.3: Modelo de datos - dominio del chat

5.2 Arquitectura y patrones de diseño

5.2.1. Backend

En este apartado se explicará la arquitectura y los patrones de diseño utilizados.

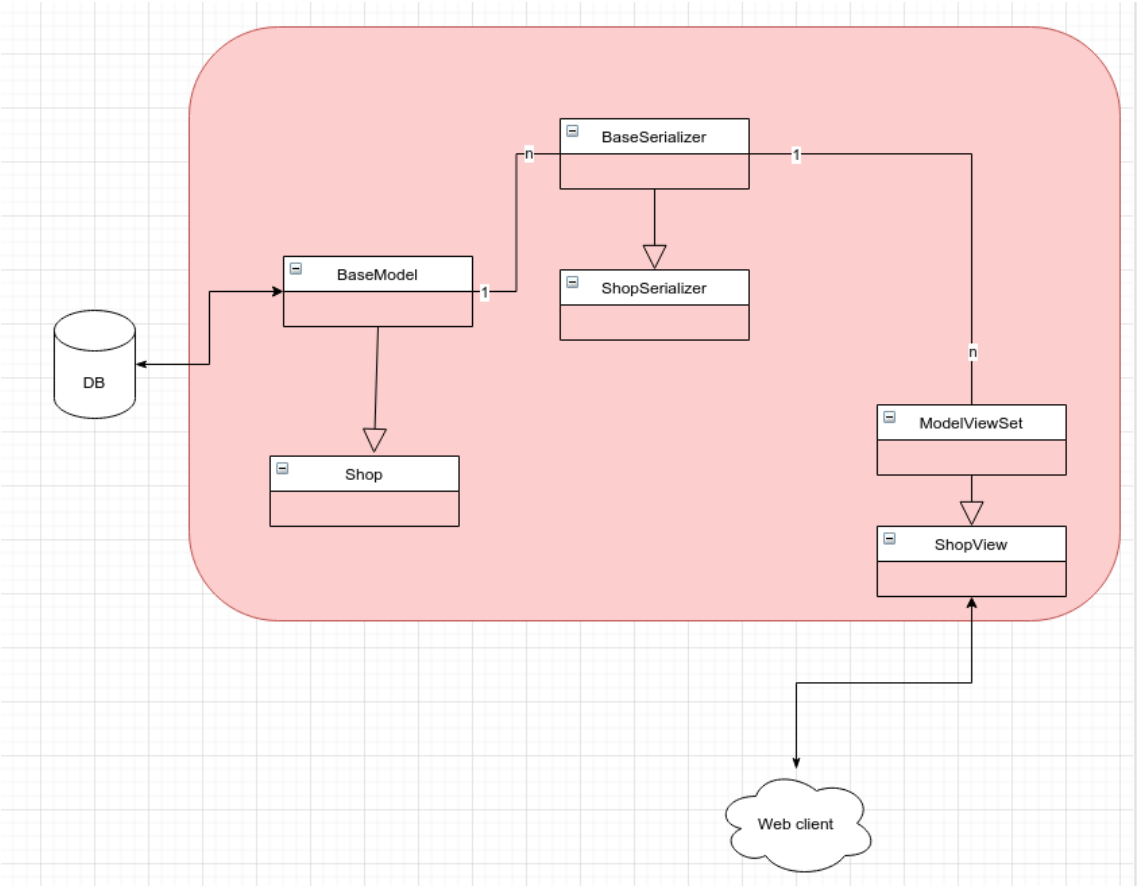

Figura 5.4: Arquitectura de la tienda

Django obliga el uso de la arquitectura Model View Template (MVT), aunque en nuestro proyecto hemos obviado uno de los componentes. En la Figura 5.4 se puede observar la aplicación de dicha arquitectura para uno de los modelos: la tienda.

Podemos observar como la vista interacciona con el cliente y el modelo interacciona con la base de datos, pero el punto de enlace entre los mismos es una clase serializadora, encargada de transformar los objetos modelo en objetos que un cliente pueda entender, en este caso, JavaScript Object Notation (JSON), así como aceptar el mismo formato como entrada y transformarlo en objetos modelo. Este patrón se replica para todos los modelos.

En la figura aparece el concepto de cliente web, y este puede ser cualquier programa que sepa establecer una comunicación Hypertext Transfer Protocol (HTTP), pero a efectos prácticos esto hace referencia a lo que hasta ahora hemos llamado frontend, el cual consume las vistas para renderizar los datos serializados.

5.2.2. Frontend

Mimetizando el apartado anterior, este sigue el mismo esquema, mostrando la arquitectura general utilizada para construir el *frontend web*.

Por la idiosincrasia del propio framework utilizado, *Svelte*, el frontend ha sido desarrollado con una arquitectura basada en componentes en mente. Esto significa que una página web cualquiera renderiza cierto número de componentes, añadiendo dinamismo, si es necesario. Para ello, creemos que el mejor diagrama para mostrar esto es el diagrama de componentes:

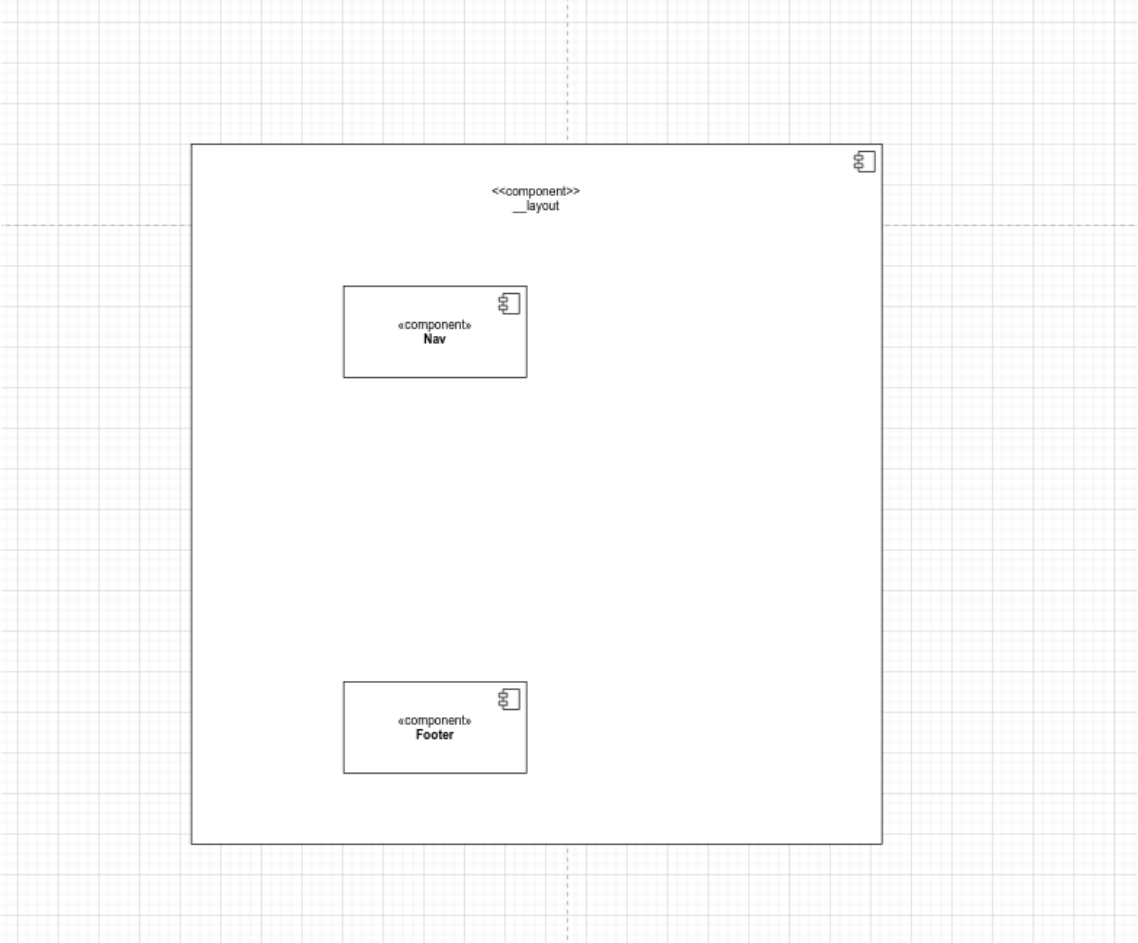

Figura 5.5: Diagrama de componentes – base

Como observamos en la Figura 5.5, se define, cualquier página renderizada por Agropop se encuentra encapsulada en el componente raíz `__layout`.

Entre los componentes `Nav` y `Footer` se encontraría otro componente que añadiese cualquier tipo de interactividad o lógica.

Tomando como ejemplo la Figura 5.6, observamos cómo la vista necesita un elemento de entrada Kiosko, el cual es inmediatamente pasado al elemento encargado de preguntar al servidor los productos del Kiosko. Una vez los tiene este carga un componente `ProductCard` por cada producto encontrado en el componente visual del kiosko.

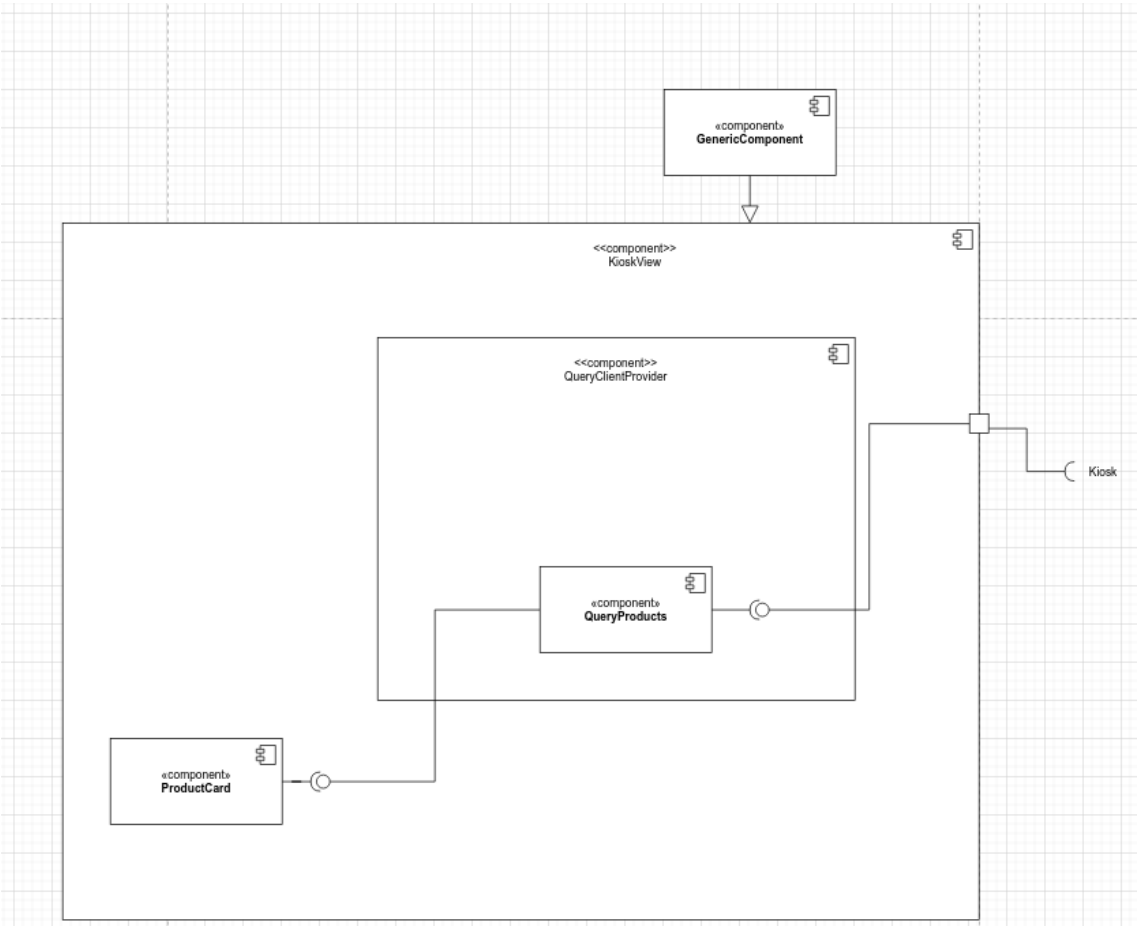

Figura 5.6: Diagrama de componentes – vista de un kiosko

CAPÍTULO 6
Desarrollo de la solución propuesta

En este apartado observaremos cómo se ha pasado de la definición abstracta anterior a una solución real con algunos ejemplos, tanto en código como en vistas.

6.1 Identificación de soluciones posibles

La idea principal de Agropop es esbozar una base para desacoplar backend y frontend; aunque esto parezca obvio en un primer momento, es interesante destacarlo porque la separación es completa: el backend puede funcionar perfectamente sin frontend al ser básicamente una Representational state stransfer (REST) API disponible para cualquier cliente que pueda hacer peticiones web.

A partir de esta decisión, las tecnologías a utilizar varian en función de las dos partes lógicas establecidas: para el backend se utiliza el lenguaje *Python* con la ayuda de *Django* y Django REST Framework (DRF)[1] mientras que en el frontend se ha decidido utilizar *Svelte* y `Typescript`

6.1.1. Por qué Python

El principal motivo por el uso de *Python* es por la experiencia que el equipo de Agropop ya tiene con el lenguaje, lo que facilita la puesta en marcha del proyecto, estableciendo de base una mayor productividad.

Por otro lado, el lenguaje ofrece otras ventajas, entre las cuales destacamos:

- Gran soporte comunitario, ya que es uno de los lenguajes más usados del mundo, según encuestas:
  - Encuesta de las tecnologías más usadas de StackOverflow en 2021[2]
  - Encuesta de las tecnologías más usadas de JetBrains en 2021[3]
- Lenguaje multiparadigma que permite utilizar orientación a objetos y programación funcional
- De código libre con una organización[4] por detrás que continuamente lo mejora

[1] Más información en https://www.django-rest-framework.org/
[2] https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language-prof
[3] https://www.jetbrains.com/es-es/lp/devecosystem-2021/
[4] la PSF o *Python Software Foundation*

- Agnosticismo a nivel de sistema operativo siempre y cuando no se utilice alguna API específica, esto es, no hay que reprogramar para poder correr un programa en diferentes sistemas operativos siempre que uno no se adentre a un nivel más bajo del que debe

Por otro lado, *Python* supone una serie de desventajas que hay que tener en cuenta, especialmente:

- La velocidad suele ser un problema cuando una aplicación escala a cierto número de usuarios
- Tiene un sistema de tipado débil, el cual recibe el nombre de *duck typing*[5]

Por qué Django y DRF

Como se ha mencionado con anterioridad, *Python* ha conseguido acumular una gran comunidad de usuarios y, consecuentemente, los problemas más comunes están implementados por dicha comunidad, varias veces. Es el caso de los *frameworks* que asisten la implementación de servicios web, habiendo un gran rango de alcances diferente al que cada uno aspira. En el caso de Agropop, se ha elegido Django porque establece rápidamente una base de desarrollo, ofreciendo: resolución de URL mediante el concepto de vista -explicado más adelante-, un ORM muy competente que permite abstraer completamente cualquier base de datos relacional, un sistema estándar de usuarios, con autenticación y todas las utilidades necesarios y, por último, un apartado web de administración extendible.

Mencionar que Django obliga a implementar una arquitectura MVT, donde *Model* hace referencia a los modelos -abstracciones de las tablas de la base de datos-, *View* hace referencia a lo que en otras arquitecturas sería conocido como capa de lógica de negocio, pues se encarga de la modificación de los datos dada una petición de un usuario. La última parte, *Template*, hace referencia a la capa de presentación y es opcional. En el caso de Agropop, hemos renunciado a utilizar esto en favor de un frontend externo que se explicará más adelante.

Además de todo esto, Django es el framework web con mayor comunidad, lo que implica que aunque la base no resuelva todos los problemas que Agropop intenta abordar, muchos de ellos están ya implementados por dicha comunidad que hay alrededor. Un ejemplo de ello, y que se explicará en profundidad más adelante, es cómo, por defecto, *Django* ofrece un sistema de autenticación basada en sesiones [6], un sistema perfectamente viable para servicios que renderizan una capa de presentación acoplada a sí mismos. Como Agropop utiliza un frontend desacoplado, esto no resulta una buena idea y es necesario un sistema de autenticación basado en token, donde el estado del usuario se guarda en el cliente. Uno de los estándares para resolver este problema es *oauth*[7], que a priori tendría que ser implementado. Sin embargo, existen múltiples módulos comunitarios que implementan el estándar. En Agropop se ha optado por utilizar *django-oauth-toolkit*[8].

No todo son ventajas para este framework, y es que al intentar albergar tanta funcionalidad de base, implementa un estilo arquitectónico monolítico, lo cual hace que pierda

[5] un sistema de tipado no fuerte basado en un dicho inglés, "*si camina como un pato y grazna como pato entonces tiene que ser un pato*", dando a entender que el simple contexto de una variable destaca su tipo

[6] la autenticación basada en sesiones dicta que la información de una sesión de usuario se guarda en la memoria del servidor, mientras que el cliente guarda un identificador de dicha sesión, transmitiendo dicho identificador en cada petición subsecuente

[7] https://oauth.net/2/

[8] https://github.com/jazzband/django-oauth-toolkit

flexibilidad, sobre todo en aspectos básicos, como por ejemplo, no poder utilizar un sistema de ficheros personalizado.

Otra desventaja es que no ofrece un estándar para definir API web, lo que nos lleva a introducir el segundo pilar tecnológico del proyecto, DRF, una librería que nos permite crear la RESTful API deseada. Existen un gran número de consideraciones y subestándares a la hora de crear una RESTful API y DRF ayuda a crear automáticamente a crear una que sea con la OpenAPI Specification (OAS), donde:

"La OpenAPI Specification (OAS) define un estándar o interfaz agnóstica a cualquier lenguaje de programación para la descripción de API HTTP, la cual permite descubrir y entender las capacidades de un servicio tanto a humanos como a ordenadores, sin tener acceso a código fuente, documentación adicional o la necesidad de inspeccionar tráfico de red. Cuando un servicio es definido adecuadamente usando OAS, un cliente puede entender e interaccionar con el servicio remoto con una cantidad mínima de lógica implementada. De manera similar a lo que las descripciones de las interfaces han hecho por los lenguajes de bajo nivel, la especificación OpenAPI elimina la necesidad de inferir a la hora de llamar a dicho servicio"(OpenAPI Specification v3.1.0)

6.1.2. Por qué Svelte(Kit)

Svelte (y *SvelteKit*) es una tecnología de componentes que nace en contraposición a las más famosas nacidas del concepto de reactividad[9] web en los últimos años. Estas tecnologías -*React*, *Vue*, etc.- implementan dicha reactividad en el navegador mediante un concepto llamado *Virtual Document Object Model (DOM)*, donde cada cambio en la implementación obliga a una serie de operaciones de diferencia entre el nuevo componente creado con los nuevos datos y el componente con los datos obsoletos, y esto tiene una serie de implicaciones.

"El *DOM virtual* ofrece mucho valor porque permite contruir *apps* sin pensar en las transiciones entre estados, con un rendimiento suficientemente bueno. Esto conlleva a un código con menos errores y más tiempo dedicado a tareas creativas y no tediosas.

Pero resulta que podemos alcanzar un modelo de programación parecido sin utilizar el *DOM Virtual* – y es ahí donde *Svelte* entra en acción. "(Rich Harris, 2018)

El crecimiento de esta técnica ha llevado a la cúspide del desarrollo web a *React*, siendo efectivamente el framework más utilizado para crear nuevas aplicaciones web. Consecuentemente, en un caso similar a *Django*, esto ha resultado en la orgánica creación de una gran comunidad alrededor, teniendo ya acceso a la implementación comunitaria de un montón de problemas que se puedan encontrar a la hora de abordar proyectos web. El problema de este hecho es que *React ha reinventado la rueda*, esto es: no es compatible con librerías *Javascript* nativas o que estén pensadas para otras tecnologías, lo que significa que gran parte de la comunidad programa para una sola tecnología sin tener en cuenta ningún tipo compatibilidad. Este hecho, a priori, podría afectar a *Svelte* -y a las demás tecnologías-, sin embargo, *Svelte* está programada de tal manera que es compatible con cualquier código nativo: es como escribir *Javascript y Hypertext Markup Language (HTML)* nativos, pero a otro nivel. Esto es porque *Svelte* no es un framework al uso, sino un compilador, con todo lo que ello implica.

Aunque *Svelte* lleve ya unos años madurando, siendo una tecnología perfectamente capaz, se ha optado, en Agropop, por utilizar no *Svelte* sino *SvelteKit*[10], que es una com-

[9] reactividad es un concepto que permite escribir estructuras de código donde lo que dicta el comportamiento es el estado de los datos, esto es: la habilidad para actualizar los componentes que el usuario ve en el instante en que la información liagada a los componentes cambia

[10] https://svelte.dev/blog/whats-the-deal-with-sveltekit

binación de tecnologías en un solo pack, que ayuda a iniciar rápidamente un proyecto web. Entre ellas, evidentemente está la propia *Svelte* pero también se combinan tecnologías para facilitar el *bundling*[11] como *Vite*. La desventaja de esta tecnología es que es aún está en fase de desarrollo, estando la versión 1.0 cerca de ser liberada.

6.2 Implementación del backend

6.2.1. Plataforma base

Como ya se ha explicado con anterioridad, la plataforma base, *Django* hace bastante fácil la iniciación de un proyecto, ofreciendo una interfaz de línea de comandos que ayuda a, entre otras cosas, añadir lo que la propia librería denomina como aplicación, y aquí hay que hacer un pequeño inciso para entender este concepto, ya que puede traer confusión.

Para *Django*, un proyecto hace referencia a un paquete que puede existir por sí mismo. En nuestro caso, Agropop en sí es un proyecto. Por otro lado, una aplicación es un módulo o paquete perteneciente al proyecto encargado de desarrollar la lógica en sí. Las aplicaciones pueden relacionarse entre sí estableciendo relaciones de dependencia. En el caso de Agropop, existen dos aplicaciones: `shop` y `chat`, siendo la primera la aplicación principal que define los modelos base para el funcionamiento del proyecto mientras que la segunda engloba toda la lógica necesaria para implementar el chat; esta última se explicará más a fondo en el siguiente capítulo.

Migraciones

Una vez uno crea una aplicación dentro de un proyecto, el siguiente paso suele ser definir el modelo de datos, las clases que interactúan con la base de datos. Por otro lado, es casi idiosincrático para un proyecto software que las clases, sobre todo las que definen una capa de abstración de las tablas de la base de datos, evolucionen a medida que el proyecto adapta nuevas necesidades, el análisis de requisitos es erróneo o cualquier otro motivo que pueda provocar un cambio. Todo esto es relevante porque, incidiendo una vez más en el concepto de ORM explicado anteriormente, la unión entre modelos *Django* y sus tablas es estática en un sentido muy problemático: si el modelo evoluciona, la tabla también tiene que hacerlo, pero si la tabla ya tiene datos, ¿cómo son estos adaptados a la nueva versión? Para esto *Django* -y otros proyectos similares- introducen las migraciones, unos script autogenerados pero modificables a posteriori que se encargan de traducir metadatos de la base de datos entre versiones.

Por ejemplo, en la versión `dc253d844e`, el modelo *Product* tiene un campo para guardar el precio base del mismo.

```
# product.py

from djmoney.models.fields import MoneyField
class Product(BaseModel):
    base_price = MoneyField(max_digits=14, decimal_places=2,
        default_currency="EUR")
```

[11] *bundling* hace referencia a la técnica usada para combinar los varios componentes de un proyecto *Javascript* y sus dependencias en un solo fichero que luego pueda ser distribuido por navegadores

Este campo viene definido por una librería externa llamada *django-money*[12], la cual ayuda a abstraer campos que trabajen con operaciones económicas. Si en algún momento se decide que este campo tiene que pasar a ser de tipo, por ejemplo, flotante o doble, habría que cambiar la anterior definición y generar los scripts de migración.

A nivel más básico, este campo encapsula otros dos: el propio valor, como flotante, y un nuevo valor de tipo cadena de caracteres donde se codifica el tipo de moneda estándar, como EUR, o USD. Por ejemplo, una representación real en base de datos puede ser la vista en la Tabla 6.1.

id	name	handler	base_price	base_price_currency
0	Banana	MyKioskBanana	0.25	EUR
1	Apple	MyKioskApple	1.36	USD

Tabla 6.1: Tabla básica de datos de producto con tipo de moneda

Vemos, como se ha mencionado, que aunque el campo es único a nivel de código, viene representado con dos columnas en la tabla: `base_price` y `base_price_currency`, siendo esta columna la que guarda el código de moneda.

Para cambiar el tipo del campo simplemente es necesario hacerlo en la definición del modelo.

```
# product.py

from djmoney.db import models
class Product(BaseModel):
    base_price = models.FloatField()
```

Después se genera el script de migración:

```
# Generated by Django 3.1.13 on 2021-06-03 11:02

from django.db import migrations, models


class Migration(migrations.Migration):

    dependencies = [
        ('shop', '0004_auto_20210903_1113'),
    ]

    operations = [
        migrations.RemoveField(
            model_name='historicalproduct',
            name='base_price_currency',
        ),
        migrations.RemoveField(
            model_name='product',
            name='base_price_currency',
        ),
        migrations.AlterField(
            model_name='historicalproduct',
            name='base_price',
            field=models.FloatField(),
```

[12] https://github.com/django-money/django-money

```
        ),
        migrations.AlterField(
            model_name='product',
            name='base_price',
            field=models.FloatField(),
        ),
    ]
```

Donde podemos observar cómo se han generado varias cambios en la lista `operations`. En especial, se puede ver en la Tabla 6.2 cómo se hace una operación `RemoveField` del campo que guardaba la opera el código que representaba la moneda.

id	name	handler	base_price
0	Banana	MyKioskBanana	0.25
1	Apple	MyKioskApple	1.36

Tabla 6.2: Tabla básica de datos de producto sin tipo de moneda

Y esto se ve reflejado en la base de datos: la columna `base_price_currency` ya no existe.

Este tipo de migraciones se tienen que planificar de manera contundente, pues en este mismo caso, aunque sencillo, se ha perdido una columna entera de datos y si se quisese volver volver a un estado anterior sería una tarea mucho más difícil, puesto que no se han guardado ningún tipo de metadatos que asocie según qué código de moneda a según qué valores de producto. La forma más sencilla de revertir este campo a`MoneyField` sería asignando un código por defecto a todas las filas.

Generador de *handlers*

Aunque a priori pueda parecer insignificante, uno de los mayores desafíos de Agropop era generar identificadores únicos para los recursos públicos que los humanos pudiesen entender. Esto, a nivel más técnico, tiene ciertas implicaciones:

- Tiene que ser parametrizable en cuanto al separador de palabras y al número de palabras: entre cuatro y diez
- Tiene que formar una frase coherente, lo que significa que sigue un estándar gramatical común del idioma elegido, en este caso, inglés. Profundizando en esto, los adjetivos tienen que preceder al sustantivo y concatenar una preposición correcta *-a* o *the-* si el número de palabras es mayor a tres
- Si el número de palabras es mayor a cuatro, los adjetivos tienen que tener un nexo *and*
- Las palabras no pueden ser autogeneradas, tienen que existir dado un diccionario real de sustantivos y adjetivos

Por ejemplo, si un usuario genera su tienda y no le asigna el nombre, por defecto se tienen que generar un *handler* parecido a *ARusticAggressiveCrowdedMilk* como caso por defecto, donde existen cuatro palabras y un valor como *TheHypnoticAndBoringCooperativePsychedelicBrother* en cuanto el número de palabras supera ese valor. Podemos observar cómo se ha insertado el nexo y la preposición ha cambiado.

```
# word_generator.py
# class constructor omitted

def generate(self, word_count=4, capitalize=True, separator="") -> str:
    if word_count < 3 or word_count > 10:
        raise ValueError("Word count must be a number between 3 and 10, like
            5")

    noun = choice(self.nouns)
    words: List[str] = []

    if word_count > 3:
        if noun[0] in self.vowels:
            words = ["an"]
        else:
            words = [choice(("a", "the"))]
    words.extend(sample(self.adjectives, k=word_count - 1))
    if word_count > 4:
        words.insert(2, "and")
    words.append(noun)

    return separator.join(words if not capitalize else map(str.title, words))
```

Teniendo en cuenta esto, la implementación en sí resulta sencilla en cuanto a números de líneas, pero resulta cierto desafío implementar tests automáticos, sobre todo porque se trata de valores pseudoaleatorios.

6.2.2. Otras dependencias

Chat

El chat se ha implementado utilizando un protocolo llamado *WebSocket*, un protocolo que permite a un cliente recibir mensajes de un servidor sin la necesidad de hacer una llamada previa, o lo que es lo mismo comunicación *full-dubplex*[13] sobre un mismo nodo Transmission Control Protocol (TCP).

Para la parte servidor, se ha utilizado la librería *channels*[14], la cual ayuda a la implementación del protocolo *WebSockets* integrándolo de manera fácil con *Django*. Esto último es extremadamente importante porque la autenticación e identificación de los usuarios que se comunican sigue siendo necesaria aunque el protocolo sea diferente, aunque este aspecto se verá en la plante cliente.

Como hemos comentado antes, la implementación se realiza creando una nueva aplicación, donde definiremos dos modelos, con sus subsecuentes serializadores y vistas, y una clase `ChatConsumer`, la cual implementa toda la lógica del propio chat.

Antes de adentrarnos en la implementación, conviene explicar un aspecto importante de *channels*, y es que utiliza la nueva librería nativa de `Python`, `asyncio`[15], la cual permite definir funciones concurrentes usando la sintaxis `async/await`:

```
# example from docs.python.org

import asyncio
```

[13] protocolo que permite la comunicación entre dos nodos en ambas direcciones
[14] https://github.com/django/channels
[15] https://docs.python.org/3/library/asyncio.html

```
    async def main():
        print('Hello ...')
        await asyncio.sleep(1)
        print('... World!')

    # Python 3.7+
    asyncio.run(main())
```

Esto es relevante porque el ORM de *Django* no es asíncrono en la versión utilizada, lo que significa que no se puede llamar, dentro de una función asíncrona, a operaciones de base de datos sin un *middleware* que garantice que la operación se ha realizado a nivel de base de datos antes de ejecutar la siguiente línea, para poder evitar condiciones de carrera. La librería *channels* ofrece una función con la que envolver operaciones ORM, `database_sync_to_sync` pero hace que todo el código quede mucho más largo y engorroso. Por ejemplo:

```
        async def _get_or_create_room(self):
            rooms = await database_sync_to_async(Room.objects.filter)(
                name=self.room_group_name
            )
            if room := await database_sync_to_async(rooms.first)():
                return room
            sent_by, sent_to = await self._parse_subprotocols()
            room = await database_sync_to_async(Room)(
                name=self.room_group_name, owned_by=sent_by, owned_by_2=sent_to
            )
            await database_sync_to_async(room.save)()
            return room
```

Esta es la función para crear o devolver una `Room` y existen tres llamadas a la base de datos: filtrado de un objeto por su nombre, creación de un objeto y guardado en la base de datos. Todas estas operaciones han tenido que ser envueltas en la función `database_sync_to_sync`

Para implementar el consumidor, primero es necesario crear una clase que herede de `AsyncWebsocketConsumer`, una superclase que *channels* ofrece con una serie de métodos a implementar para poder establecer la comunicación

```
class ChatConsumer(AsyncWebsocketConsumer):
    room_name: str
    room_group_name: str
    room: Room

    # some methods are omitted [...]

    async def connect(self):

        if not await \
            self.is_authenticated(access_token=self.scope["subprotocols"][0]):
            await self.close()
            return

        self.room_name = self.scope["url_route"]["kwargs"]["room_name"]
        self.room_group_name = f"chat_{self.room_name}"
        self.room = await self._get_or_create_room()
```

```
        # Join room group
        await self.channel_layer.group_add(self.room_group_name,
            self.channel_name)
        await self.accept()

    async def disconnect(self, close_code):
        await self.channel_layer.group_discard(self.room_group_name,
            self.channel_name)

    async def receive(self, text_data=None, bytes_data=None):
        deserialized_data = json.loads(text_data)
        message = deserialized_data["message"]

        sent_by, sent_to = await self._parse_subprotocols()
        await self.create_chat_message(message, sent_to, sent_by)
        # Send message to room group
        await self.channel_layer.group_send(
            self.room_group_name,
            {
                "type": "chat_message",
                "sent_by": {"id": sent_by.id, "handler": sent_by.handler},
                "sent_to": {"id": sent_to.id, "handler": sent_to.handler},
                "message": message,
            },
        )

    async def chat_message(self, event):
        message = event["message"]
        await self.send(text_data=json.dumps({"message": message}))
```

El método principal es *connect* y es llamado en cuanto la comunicación *WebSocket* es establecida por primera vez, actuando así como un constructor que inicializa los atributos sin valor definidos al principio de la clase: `room_name`, `room_group_name` y `room`. Pero antes de realizar esto es necesario comprobar que la comunicación está correctamente autenticada: el cliente tiene que añadir el token de acceso en el mensaje de conexión para hacer esta comprobación. Este se encuentra en `self.scope["subprotocols"][0]`. Si el token es incorrecto o no existe, la comunicación es interrumpida.

En caso contrario, se extrae el nombre de la sala y los usuarios: el usuario autenticado a través del token anterior y el *handler* del otro usuario en el siguiente elemento de la lista de subprotocols anterior: `self.scope["subprotocols"][1]`. Esto es transparente a este método porque los usuarios son encapsulados por la propia `Room`, por lo que son inicializados cuando se llama a `self._get_or_create_room`.

El método para gestionar el envio de los propios mensajes es `receive`, el cual espera recibir datos JSON, los deserializa a un rama y del mismo extrae el mensaje y los *handlers* del emisor y receptor, con los que se crea un mensaje en la base de datos y se reenvia a los otros clientes para que puedan establecer que el mensaje ha sido registrado en el servidor central.

Mencionar que algunos métodos mencionados son omitidos del código mostrado porque son simples llamadas a la base de datos o conversiones y serializaciones de datos.

Geolocalización

La implementación de la geolocalización puede parecer un aspecto muy complicado, y lo es si se parte de la nada. Sin embargo, nuevamente *Django* ofrece una interfaz muy

cómoda para implementar estructuras tanto sencillas como complejas de tratamiento de datos espaciales, con la ayuda del paquete *GeoDjango*, que utiliza *PostGIS* por debajo para persistir los datos en la base de datos. Este último obliga a que Agropop solo pueda trabajar con bases de datos *PostgreSQL*.

A nivel conceptual todo esto se basa en un modelo llamado GIS, encargado de definir y analizar datos geográficos. Este modelo sirve para un gran número de representaciones además de un punto en un mapa pero esto no corresponde al marco de este trabajo. Aún así es ideal para el problema presente.

La puesta en marcha necesita ciertas dependencias que se verán explicadas en el Sección 6.5.

```
from django.contrib.gis.db import models as gis_models


class Kiosk(BaseModel):
    # [...]

    location = gis_models.PointField(null=True)
```

Una vez esas dependencias son resueltas, añadir un punto geográfico a un modelo es tan simple como importar `PointField` de los modelos GIS y asignarlo a un atributo. Este es un campo complejo que por defecto *Django* no sabe serializar a un formato accesible, retornando una simple cadena de caracteres que muestra la latitud y longitud. Un ejemplo: `'SRID=4326;POINT (-0.4404187202453614 39.30242456041487)'`. Este formato no es muy óptimo para que sea tratado por un cliente que trabaje con JSON, por lo que es necesario añadir tratamiento extra en el serializador. Esto se puede hacer de manera manual o utilizando una librería que la comunidad pone a disposición de cualquiera, en este caso, el paquete *drf-extra-fields*[16].

```
from drf_extra_fields.geo_fields import PointField


class KioskSerializer(BaseSerializer):
    location = PointField()
```

Al sobreescribir la definición del atributo `location` del serializador al nuevo atributo *PointField*, y consultar nuevamente la información de un kiosko, lo que recibimos es:

```
    "location": {
      "latitude": 39.30242456041487,
      "longitude": -0.4404187202453614
    },
```

Observamos que es la misma información que vimos anteriormente pero en un formato mucho más accesible.

[16] https://github.com/Hipo/drf-extra-fields

6.3 Implementación del frontend Frontend

6.3.1. Plataforma base

DOM

El DOM es la estructura y representación jerárquica básica que compone la web. Todo lo que ocurre en una página web en un navegador dado ocurre manipulando esta estructura.

Como se ha mencionado, el DOM ofrece una estructura jerárquica y esto se observa con solo escribir un número mínimo de líneas HTML, donde:

```
<!DOCTYPE html>
    <head>
    </head>
    <body>
        <p>Hey, this is the document's body!</p>
    </body>
</html>
```

Podemos observar cómo del elemento padre, `html`, cuelgan los siguientes elementos `head` y `body`, este último teniendo a su vez un elemento hijo que describe un párrafo. El elemento `html` es el elemento de mayor jerarquía al que tenemos acceso en el lenguaje de marcado, pero el DOM ofrece una lista de otros elementos que están por encima de este: en especial, `document`, y `object`, los cuales pueden ser accedidos desde un script y manipular los hijos de los mismo.

Esta explicación de las bases de la web es necesaria para entender correctamente los siguientes conceptos que *SvelteKit* ofrece.

SvelteKit

SvelteKit busca unificar en un solo paquete varias tecnologías que hasta el inicio de su existencia estaban separadas en otros paquetes. Una de estas tecnologías era *Sapper*, utilizada para construir aplicaciones web utilizando prácticas modernas como el concepto de Server-side rendering (SSR), donde una aplicación web no manipula el DOM en el navegador sino en el servidor, donde se transforma en cadenas HTML y son devueltas con el estado modificado al cliente en un proceso llamado *hidratación*. Esto tiene ciertas implicaciones y es que hay que programar con este proceso en mente en todo momento.

Una línea de código que utiliza operaciones que llamen al objeto *document* cuando el código se está ejecutando en el servidor no se podrá ejecutar porque ese objeto pertenece al navegador, es el objeto necesario para modificar el DOM, por lo que la librería que implementa esto tiene que ofrecer algún mecanismo para comprobar en qué nodo se está ejecutando qué código.

Svelte resuelve esto con unas funciones de ciclo de vida, mimetizando otras tecnologías como *Unity* o *Android*. Una de ellas es la función *onMount*, la cual se ejecuta cuando el código llega al navegador.

Otras implicaciones negativas de este tipo de tecnologías es que ahora gran parte del peso de la ejecución de una aplicación de navegador cae en los hombros del servidor.

Por otro lado, el renderizado en la parte del servidor es útil en dos aspectos: mejor Search Engine Optimization (SEO) porque los indexadores tienen acceso a una página

web renderizada al 100 % y el segundo aspecto, ligado a este mismo concepto, el contenido se carga antes por el mismo principio: el navegador no es quien realiza las operaciones dinámicas. Por otro lado, otros inconvenientes aparecen, como el explicado anteriormente.

SvelteKit elimina *Sapper* de la ecuación para tener un SSR integrado.

Modelo de datos y deserialización

Otro aspecto importante del desarrollo del cliente web es el añadido de un tipado estático y fuerte a Javascript, una tarea que ha sido resuelta por *Typescript* en los últimos años. Esto es importante, sobre todo, para tener un modelo de datos consistente con el modelo de datos del servidor.

Para ello se necesita resolver dos problemas: cómo representar los modelos y cómo deserializar JSON a esas representaciones.

La primera pregunta se contesta, a priori, fácilmente: *dataclasses*, clases que tienen ciertos atributos y métodos que las orientan a albergar datos, sobre todo operaciones aritméticas y de comparación. Este concepto, aunque no nuevo, no está adaptado en la librería estándar de *Typescript* por lo que hemos de utilizar una librería externa[17]

Como respuesta a la segunda pregunta, Agropop utiliza la librería *class-converter*[18].

```
// base.ts
import Record from 'dataclass';
import { property } from 'class-converter';

export abstract class NamelessBase extends Record<unknown> {
   @property()
   id: number;
}

export abstract class Base extends NamelessBase {
   @property()
   name: string;
}

export abstract class AvatarBase extends Base {
   @property('avatar', null, true)
   avatar?: string;
}

// kiosk.ts
import { AvatarBase } from './base';
import { Shop } from './shop';
import { property, typed } from 'class-converter';
import { Product } from './product';
import { Location } from './location';

export class Kiosk extends AvatarBase {
   @property('shop', Shop)
   shop: Shop;

   @property()
   description: string;
```

[17] https://dataclass.js.org/
[18] https://github.com/zquancai/class-converter

```
    products?: Array<Product>;

    @property('products')
    productIds: Array<number>;

    @property()
    handler: string;

    @property('is_showcase')
    isShowcase: string;

    @property('location', Location, true)
    location?: Location;
}
```

Podemos observar cómo para definir el mismo modelo que tenemos en la aplicación *Django* se utilizan las librerías mencionadas. *Kiosk* hereda de *AvatarBase* que a su vez encapsula varios niveles de clases hasta llegar a *Record*, que es la clase base que convierte una clase en una *dataclass*.

Por otro lado, encima de todos los atributos podemos observar el decorador *property*, el cual le indica a la librería de deserialización que se trata de un atributo que tiene que ser convertido desde JSON. Para muchos de ellos la conversión es automática mientras sea un tipo simple -entero, doble, flotante, etc...-, no nulo y el nombre del atributo coincida con el nombre del atributo en la cadena JSON. Sin embargo, esto no siempre es el caso porque:

- El estándar de nomeclatura de variables y atributos en Python es *snake_case*, lo que singifica que si la variable tiene más de una palabra, esta tiene que ser separada por el caracter '_', todo en minúscula. Por otro lado, Typescript dicta lo contrario, hay que utilizar el estándar *camelCase*, donde si un atributo es compuesto por varias palabras, las que vienen después de la primera tiene que tener su primera letra en mayúscula. Por eso el atributo *isShowcase* define en su decorador la cadena *is_showcase*
- Los objetos que pueden ser nulos tienen que definir su nulabilidad, tanto a nivel de Typescript como a nivel decorador – el último atributo de

  `@property('location', Location, true)`

  define si un atributo es nulo -*true*- o no -*false*.

Otro problema que nos encontramos es la recursión de la deserialización, esto es, hasta qué nivel de recursión de subdatos podemos llegar a la hora tanto de pedirle datos al servidor para instanciar un objeto. A nivel abstracto, un kiosko tiene una lista de productos; si esa lista de productos es, hipotéticamente, de varios millones, no resulta viable hacer que el servidor devuelva todos los datos de cada producto de dicha lista de productos en una sola llamada, sobre todo si los productos a su vez implementan otros atributos complejos.

Por defecto, a la hora de serializar objetos completos, *Django* devuelve solo el identificador, en este caso, el atributo *handler*.

```
{
  "id": 5801,
  "name": "Shop121's 0 kiosk",
```

```
  "is_showcase": true,
  "description": "foobar"
  "products": [
    shop_121_kiosk_0_product_0,
    shop_121_kiosk_0_product_1,
    shop_121_kiosk_0_product_2,
    shop_121_kiosk_0_product_3,
    shop_121_kiosk_0_product_4,
    shop_121_kiosk_0_product_5,
  ],
  "shop": shop_121,
  "location": {
    "latitude": 39.30242456041487,
    "longitude": -0.4404187202453614
  },
  "handler": "shop_0_kiosk_0",
  "avatar": null
}
```

Para convertir este JSON a un objeto tipo *Kiosk* habrá que hacer llamadas separadas para el modelo *Shop* con handler *shop_121* y para la lista de productos. Una vez se hacen estas llamadas, se pueden asignar a sus atributos correspondiente. Sin embargo, el problema anterior no tiene que ocurrir para atributos que no son listas, mientras estos a su vez sigan la misma política. En el ejemplo anterior, es el caso del atributo *shop*.

Se puede pedir al servidor que expanda ciertos atributos complejos en la propia llamada. Para recibir el JSON anterior se ha llamado al endpoint `/kiosks/shop_0_kiosk_0/` pero si añadimos ciertos parámetros podemos expandir el atributo necesario, llamado a `/kiosks/shop_0_kiosk_0/?expand=shop`

```
{
  "id": 5801,
  "name": "Shop121's 0 kiosk",
  "is_showcase": true,
  "description": "foobar"
  "products": [
    shop_121_kiosk_0_product_0,
    shop_121_kiosk_0_product_1,
    shop_121_kiosk_0_product_2,
    shop_121_kiosk_0_product_3,
    shop_121_kiosk_0_product_4,
    shop_121_kiosk_0_product_5,
  ],
  "shop": {
    "name": "Shop 121",
    "user": 123,
    "description": "foobar",
    "handler": "shop_121",
    "avatar": null
  },
  "location": {
    "latitude": 39.30242456041487,
    "longitude": -0.4404187202453614
  },
  "handler": "shop_0_kiosk_0",
  "avatar": null
}
```

Rutas

Un aspecto muy importante que Svelte resuelve de manera más fácil que otros proyectos similares es la gestión de las rutas o *endpoints*. Mientras otras librerías resuelven este problema obligando a la definición de las rutas explícitamente a la hora de definir componentes -siendo este el caso de *Vue-*, *Svelte* aplica un sistema basado en ficheros, lo que significa que todo fichero con la extensión *.svelte* que se defina debajo de una carpeta designada llamada *routes* es automáticamente convertido a una ruta con el mismo nombre.

Estas es el sistema de ficheros actual:

```
routes
├── chat.svelte
├── index.svelte
├── kiosks
│   └── [kiosk].svelte
├── __layout.svelte
├── my
│   ├── preferences.svelte
│   └── shop.svelte
├── products
│   └── [product].svelte
├── shops
├── shops.svelte
│   ├── index.svelte
│   └── __layout.svelte .3 [shop].svelte
├── signin.svelte
├── signup.svelte
└── users
    └── [user].svelte
```

De este sistema de ficheros se pueden inferir las siguientes rutas:

- */chat/*
- */*
- */kiosks/<handler de un kiosko>*
- */my/preferences/*
- */my/shop/*
- */products/<handler de un producto>*
- */shops/*
- */shops/<handler de una tienda*
- */signin/*
- */signup/*
- */users/<handler de un usuario/*

Las rutas dinámicas, por ejemplo, `/shops/SomeShopHandler/`, donde uno de los atributos no puede ser fijo, en este caso, el *handler* de una tienda, se tienen que codificar en el

nombre del fichero utilizando la plantilla `[un nombre descriptivo].svelte`. El nombre dentro de los caracteres '[]' tiene que ser sencillo y descriptivo porque, cuando ese fichero sea llamado, una variable con el mismo nombre será inicializada para poder tener acceso al valor, que en este caso es *SomeShopHandler*.

Siguiendo con el ejemplo anterior, el fichero `[shop].svelte` tiene que mostrar los datos de una tienda dado su handler. Esto se realiza a nivel del servidor del cliente según se explicó anteriormente -SSR. Para ello hay que implementar una función llamada `load`

```
// [shop].svelte

 <script context='module' lang='ts'>
   import { Shop } from '../../api/models/shop';
   import { toClass, toClasses } from 'class-converter';
   import { simpleApiClient } from '../../api/api';

   export async function load({page,fetch}: Promise<unknown>): Promise<{ props:
       { shop: Shop } }> {
      const { data } = await simpleApiClient.get(
         `/shops/${page.params.shop}/?expand=user.user.user/`
      );
      return {
         props: {
            shop: toClass(data, Shop)
         }
      };
   }
</script>
```

Podemos ver que la función recibe dos parámetros, `page` y `fetch`. El primero tiene ciertos metadatos, entre ellos, el nombre de la tienda que necesitasmos, mientras que el segundo parámetro es una función 'fetch' especial utilizada para realizar llamadas HTTP.

Podemos ver en la primera línea de dentro de la función como se realiza una llamada al servidor *Django*, asignando el nombre de la tienda a la ruta que este tiene que recibir. Utilizando el ejemplo anterior, si resolviésemos esa ruta, quedaría algo parecido a `/shops/SomeShopHandler/?expand=user.user.user/`.

Volviendo a la definición del método, podemos observar que este tiene que devolver una promesa que encapsula un objeto *props* que contiene un objeto `shop` del tipo `Shop`, y es por lo que se hace la llamada anterior. Se inicializa un objeto tienda y se devuelve, quedando disponible al resto del código.

Svelte Query

Cargar datos del servicio, sobre todo cuando entran en cuestión parámetros como paginación y renderizado de componentes, resulta una tarea difícil. Es por eso por lo que hemos optado por una librería que abstrae este problema de una manera descrita por los propios desarrolladores como "dogmática".

Pero antes de adentrarnos en esto, vamos a explicar qué estrategia de paginación se ha elegido para el servidor.

Cuando hacemos una llamada a una ruta que devuelve una lista de datos, esta llamada también devuelve ciertos metadatos. A modo de ejemplo, utilizaremos la ruta */products/*, la cual, a priori, devuelve:

```
{
```

```
  "count": 37500,
  "next": "http://localhost:8000/api/products/?page=2",
  "previous": null,
  "results": [
    {
      "name": "Kiosk0's 0 product",
      "base_price": "0.00",
      "handler": "kiosk_0_0_product_0",
      "description": "",
      "avatar": null,
      "gallery": null
    },
    {
      "name": "Kiosk0's 1 product",
      "base_price": "0.00",
      "handler": "kiosk_0_0_product_1",
      "description": "",
      "avatar": null,
      "gallery": null
    },
    ...
    ]
}
```

Aunque existan 37500 productos, el servidor no va a devolver ese tamaño de datos en una sola llamada, y pagina sus datos de diez en diez, dejando, en cada llamada, una URL a la que llamar para recibir los siguientes datos, o los anteriores, si procede.

Hay que tener lo anterior en cuenta para trabajar con la librería de búsqueda de datos mencionada. Esta librería es *Svelte Query*[19]

Se puede hacer todo-en-uno, pero por recomendaciones de la propia librería, es mejor crear un componente por cada necesidad que tiene la misma. El principal componente es el de una propia búsqueda, por lo que hay que definir un fichero *QueryProducts.svelte*

```
// QueryProducts.svelte

<script>
   import { useInfiniteQuery } from '@sveltestack/svelte-query';
   import { apiClient } from '../../api/api';
   import { toClass } from 'class-converter';
   import ProductCard from '$lib/cards/ProductCard.svelte';
   import { Product } from '../../api/models/product';

   export let kiosk;

   const fetchProducts = async ({ pageParam = 1 }) => {
      const { data } = await apiClient().get(
         `/products/?page=${pageParam}&id__in=${kiosk.productIds.join(',')}`
      );
      return data;
   };

   const queryOptions = {
      queryKey: 'products',
      queryFn: fetchProducts,
      //@ts-ignore
```

[19] https://github.com/SvelteStack/svelte-query

```
      getNextPageParam: (lastGroup) => {
         if (lastGroup.next == null) return undefined;
         const next = lastGroup.next.split('?')[1];
         const paramQuery = new URLSearchParams(next);
         const nextPage = paramQuery.get('page');
         return nextPage || undefined;
      }
   };

   const queryResult = useInfiniteQuery(queryOptions);
</script>
```

Este caso es un poco más complejo porque no se buscan todos los productos de la base de datos sino los que pertenecen a un kiosko, de ahi la primera línea después de las importaciones, que define que cualquier otro componente que invoque a este, tendrá que pasarle un objeto tipo `Kiosk`. Por ejemplo:

```
<section class='section'>
    <QueryClientProvider>
        <QueryProducts kiosk={kiosk} />
    </QueryClientProvider>
</section>
```

Una vez definido lo anterior, lo primero es definir una función dinámica[20] que busque los productos deseados. Esta función es la función anónima asignada a la variable `fetchProducts`, la cual recibe un parámetro llamado `pageParam`, el cual indica qué página de datos hay que pedirle al servidor.

El componente no llama directamente a esta función, pues se tiene que asignar a una lista de opciones que la librería necesita, en este caso, `queryOptions`. La última opción de dicha lista es una función anónima que, dada una búsqueda anterior, transforma el campo de siguiente página en un número para poder pasárselo de nuevo a la función de búsqueda. Por ejemplo, siguiendo el ejemplo anterior de los productos, donde `next` tenía como valor `http://localhost:8000/api/products/?page=2`, si asumimos como búsqueda anterior, la función anónima sacara el número '2' de la URL para pasársela a la nueva búsqueda y cargar así la segunda página.

Por último, se utiliza una función de *Svelte Query* llamada `useInfiniteQuery`, la cual llama de manera infinita y por pasos a la función de búsqueda en tanto exista siguiente página. Esta es la función más óptima que la librería ofrece para hacer una carga de datos con un botón de *Cargar más*.

Podemos ver esto en práctica en las figuras Figura 6.1 y Figura 6.2

Otro aspecto muy importante de *Svelte Query* es que resuelve de manera totalmente transparente para el programador problemas de conectividad o sincronía con el servidor, esto es: continuamente interacciona con el servidor para ver si, aunque momentos atrás comprobase que no hay más datos por cargar, en este instante hay, comprobando así si este sigue respondiendo correctamente.

[20] en este caso, dinámica hace referencia al hecho de que tiene que tener en cuenta parámetros que pueden cambiar, como el número de página

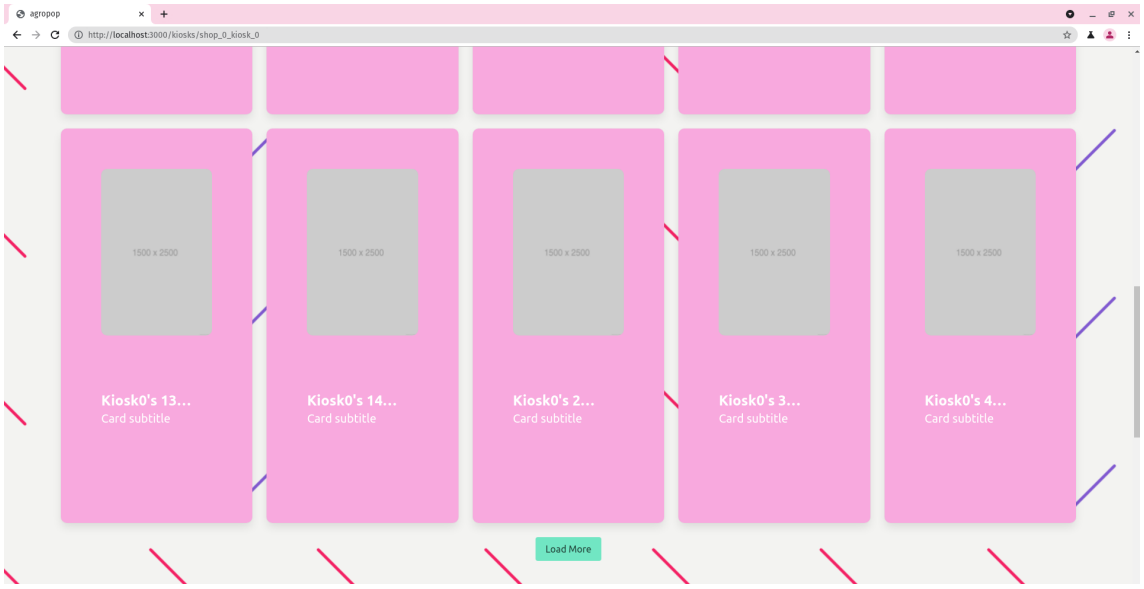

Figura 6.1: Carga infinita de productos

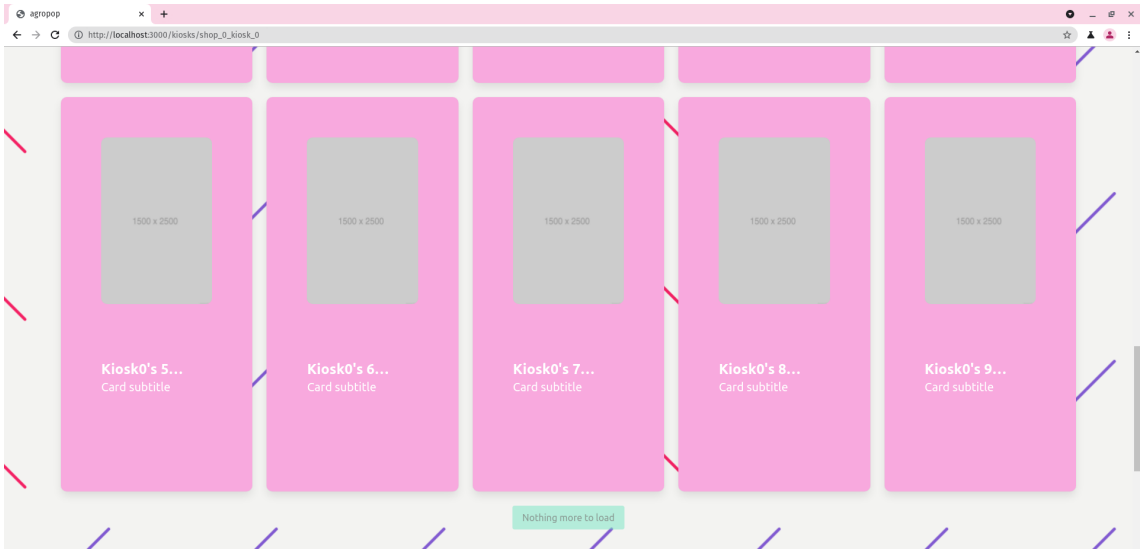

Figura 6.2: Carga infinita de productos – sin productos por cargar

6.3.2. *Framework CSS: Bulma*

Se ha elegido *Bulma*[21] como base para la confección visual de la aplicación web. Bulma ofrece una gran cantidad de componentes con estilos a la vez predefinidos y a la vez customizables, fácilmente aplicables, un sistema de grid sencillo y responsivo a cambios de tamaño de pantalla.

Por defecto, bulma define un sistema de hasta doce columnas, aunque este número puede cambiar ajustando el tamaño de las mismas: a mayor tamaño de columna, menos número, pudiendo ajustar así dinámica y fácilmente el tamaño de los subcomponentes. Esto es esencial para tener, desde el primer momento y sin necesidad de aplicar otro cambio, responsividad para, por ejemplo, pantallas más pequeñas.

[21] https://bulma.io

Figura 6.3: Pantalla de un kiosko – vista del menú superior y descripción del kiosko

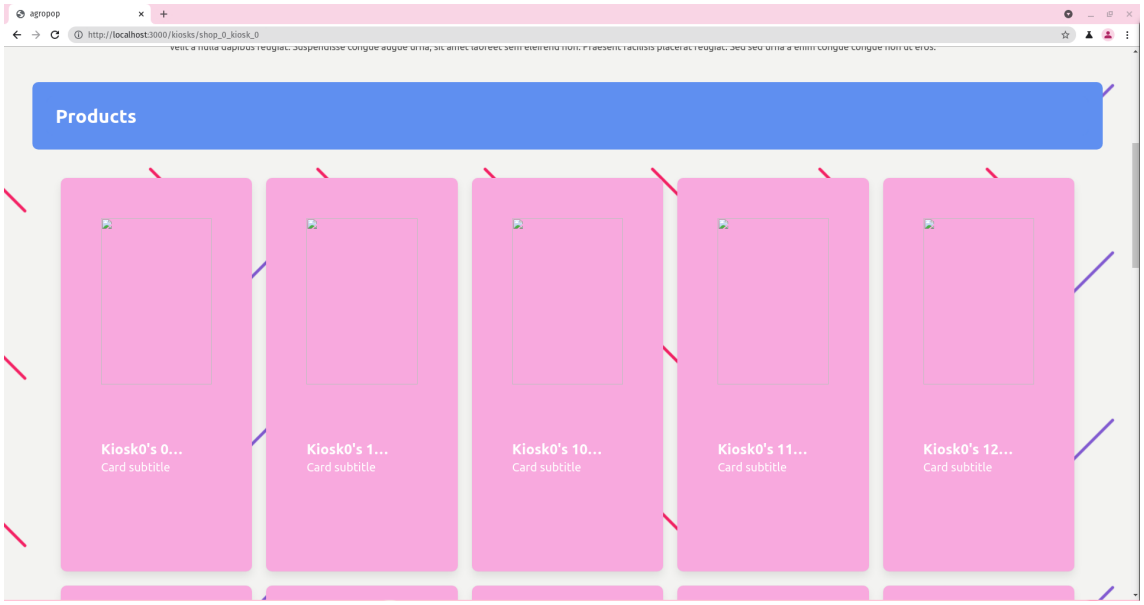

Figura 6.4: Pantalla de un kiosko – vista de los productos

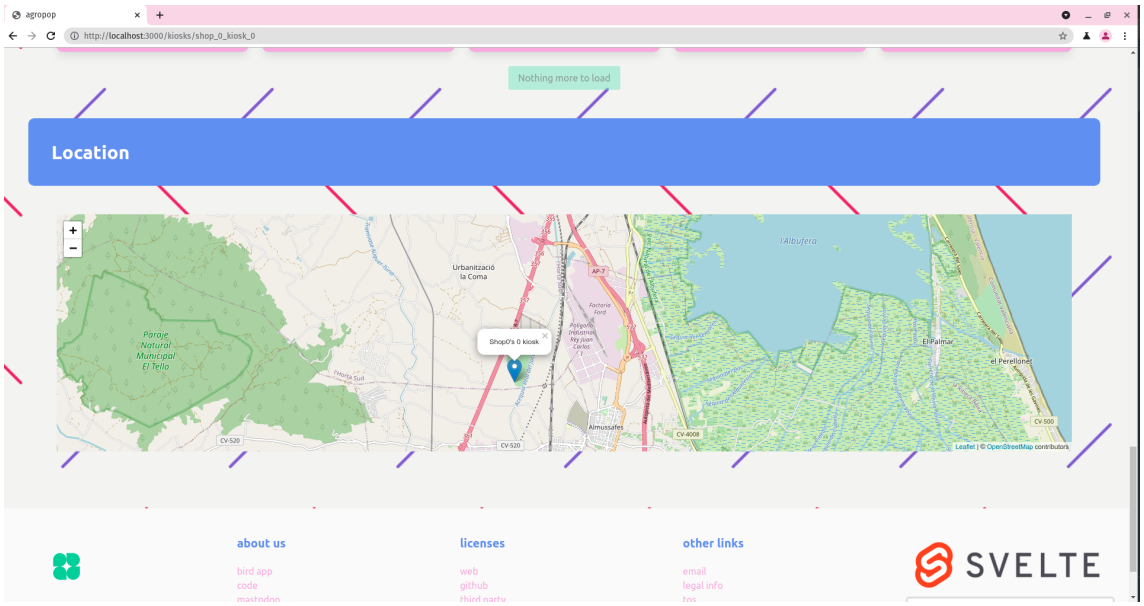

Figura 6.5: Pantalla de un kiosko – vista del mapa y footer

Podemos observar en las figuras Figura 6.3, Figura 6.4 y Figura 6.5, cómo se ve un kiosko en una pantalla normal normal de ordenador.

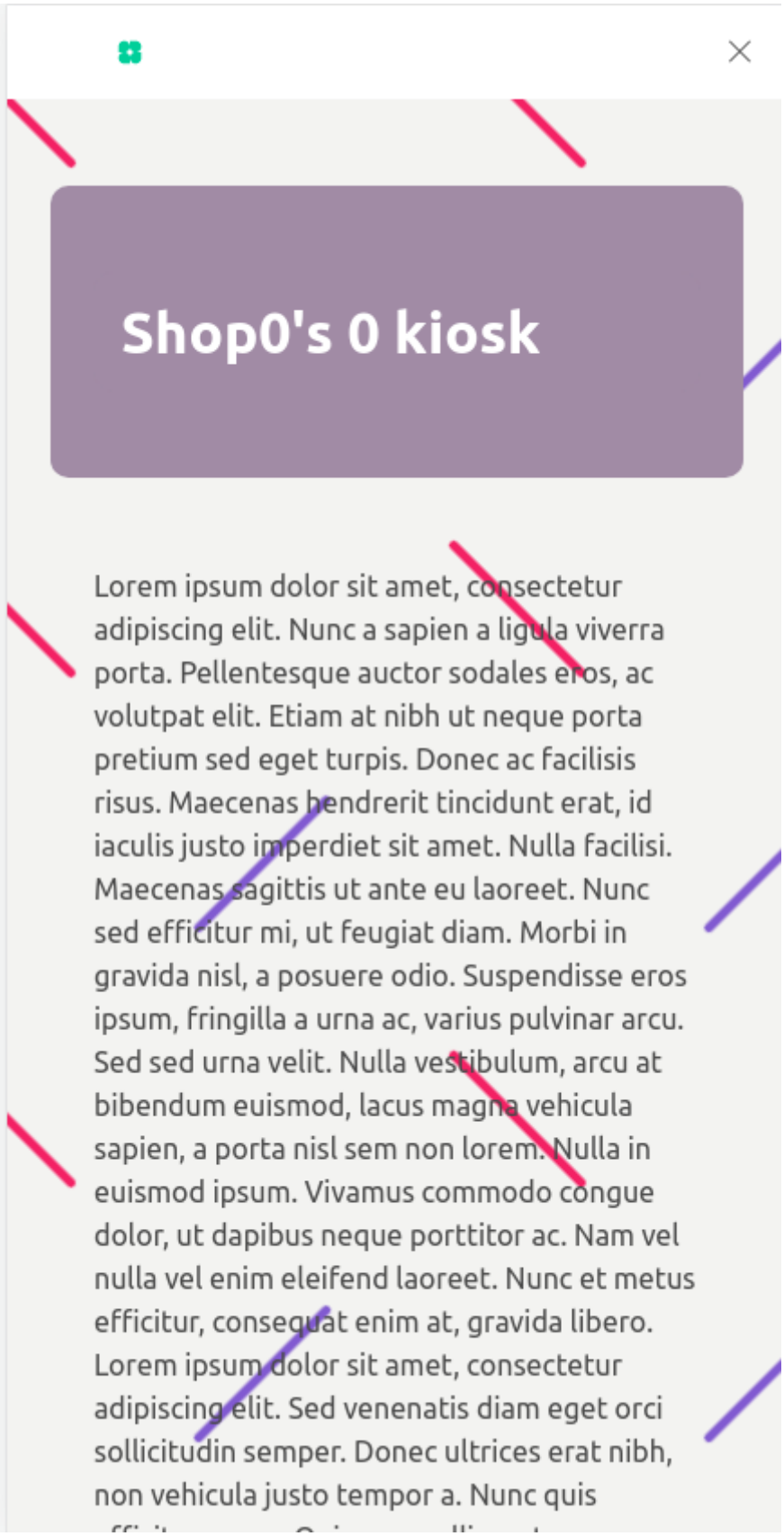

Figura 6.6: Pantalla de un kiosko con tamaño móvil – vista del menú superior

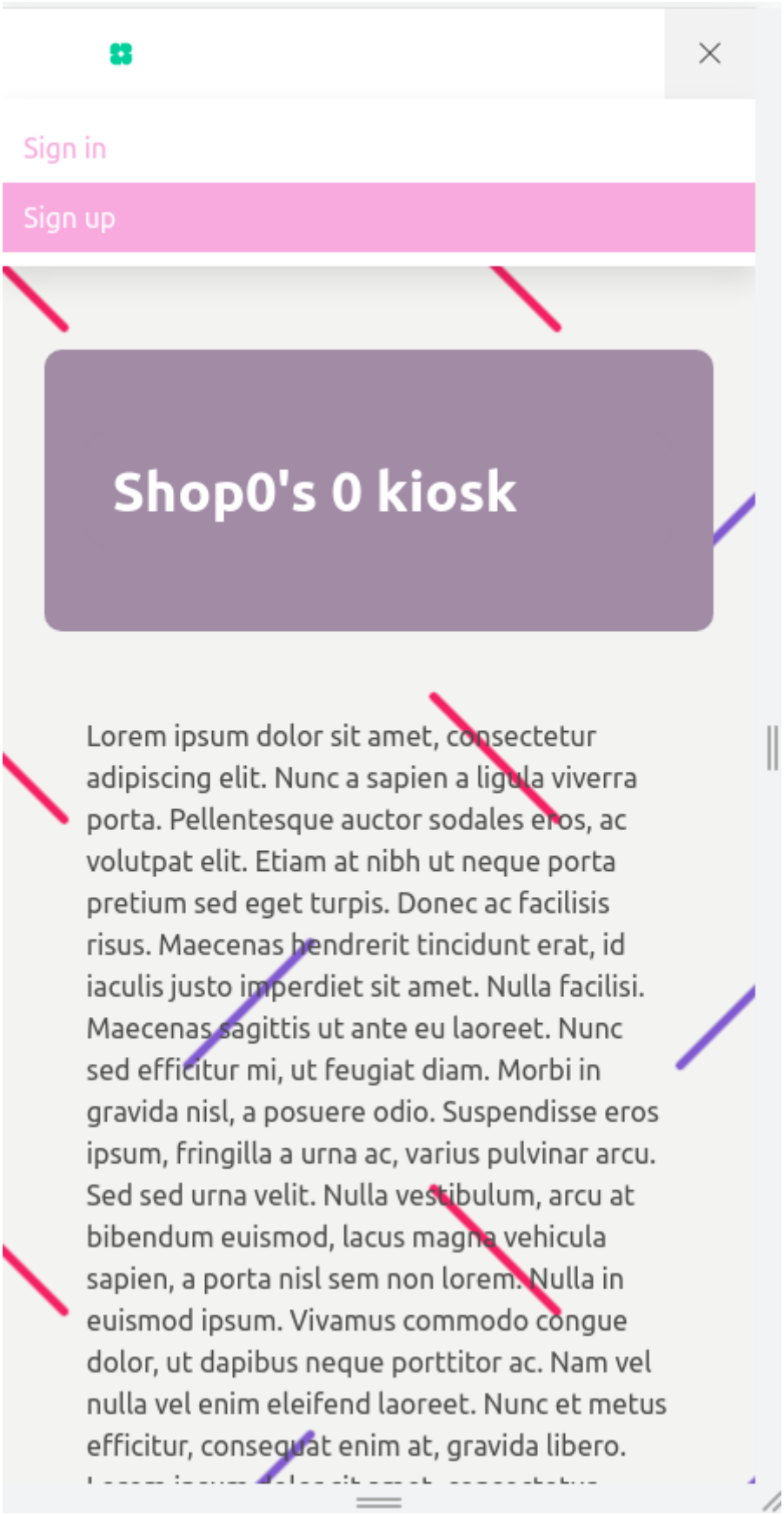

Figura 6.7: Pantalla de un kiosko con tamaño móvil – vista del menú superior abierto

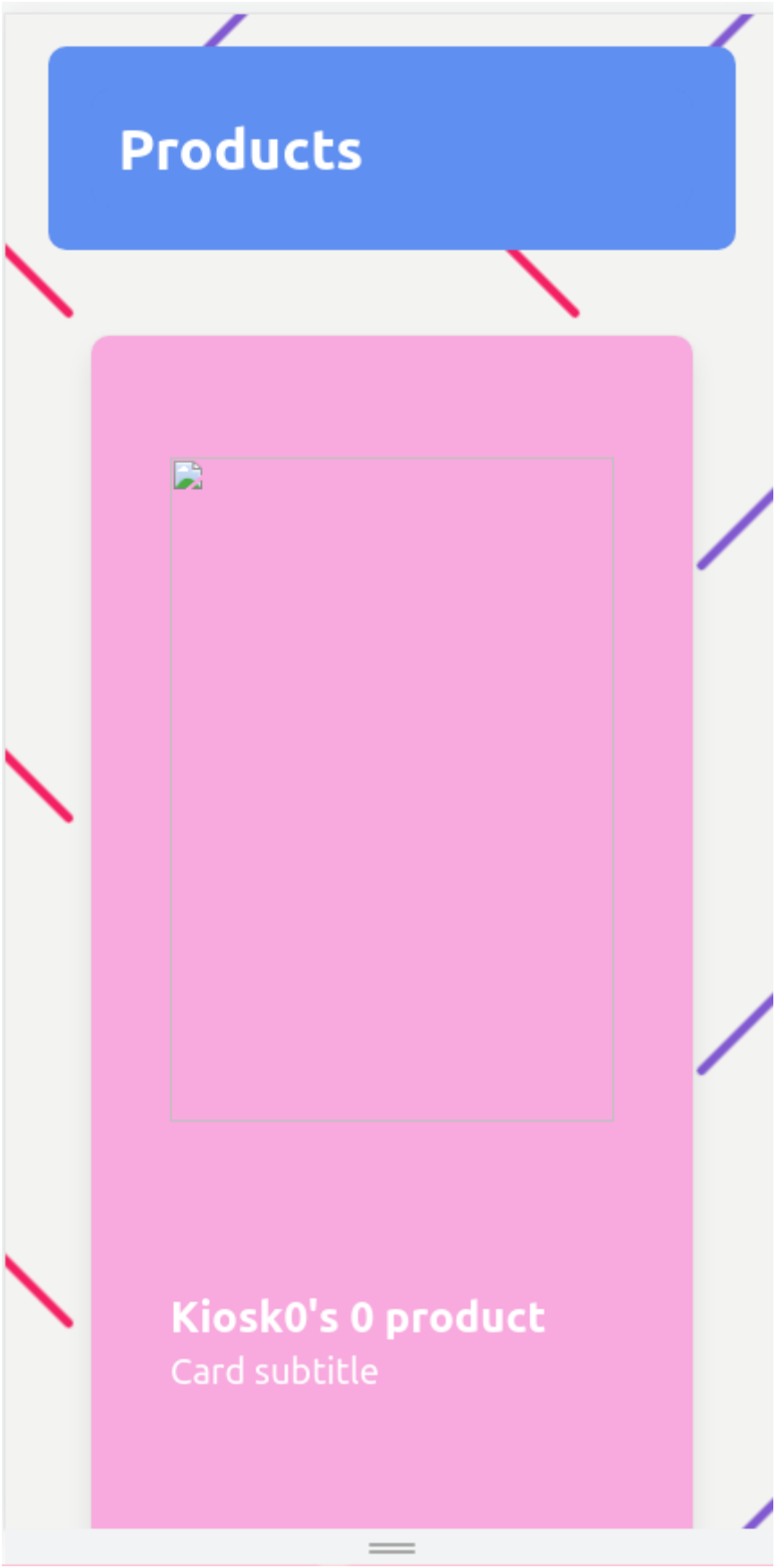

Figura 6.8: Pantalla de un kiosko con tamaño móvil – vista de productos

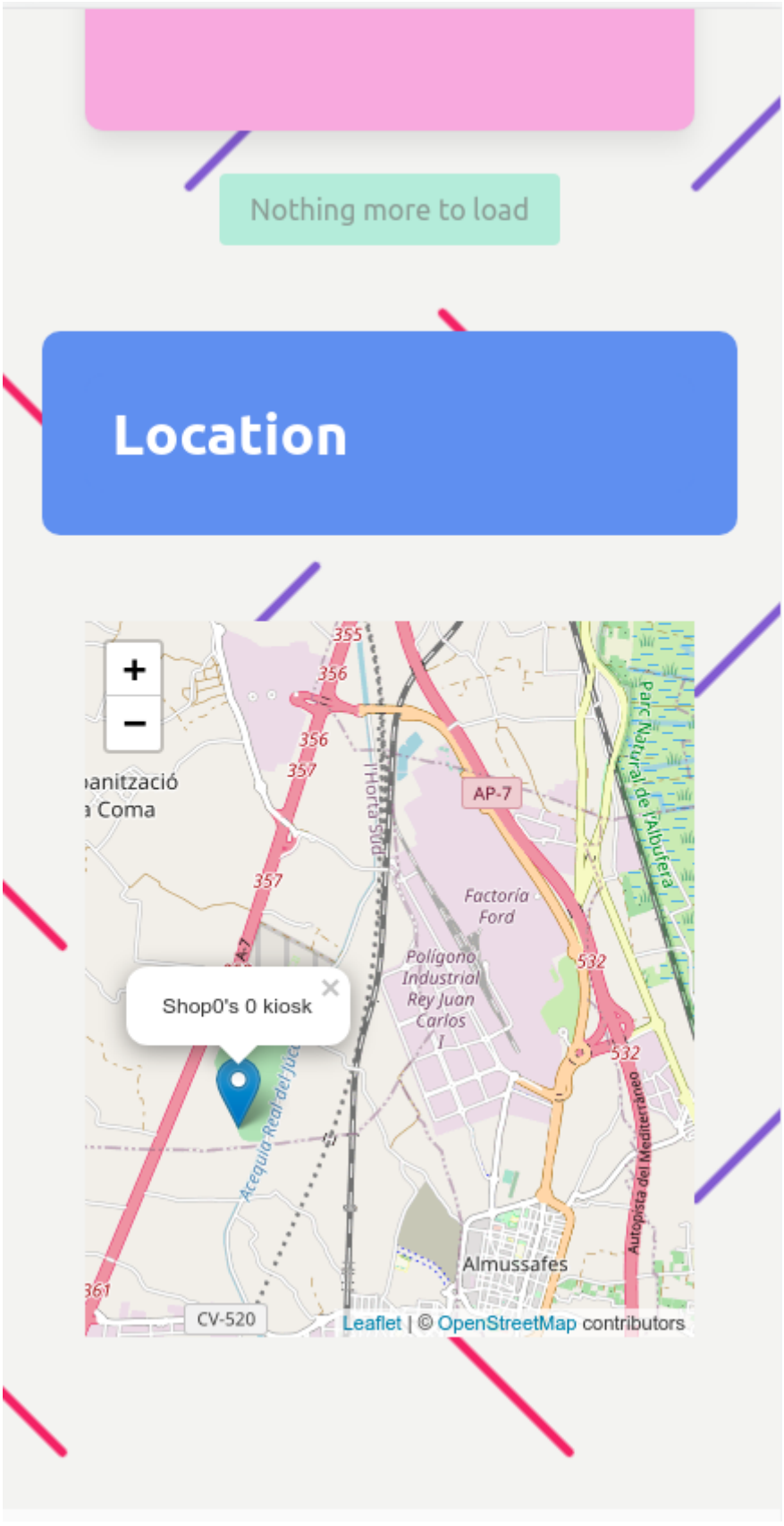

Figura 6.9: Pantalla de un kiosko con tamaño móvil – vista del mapa

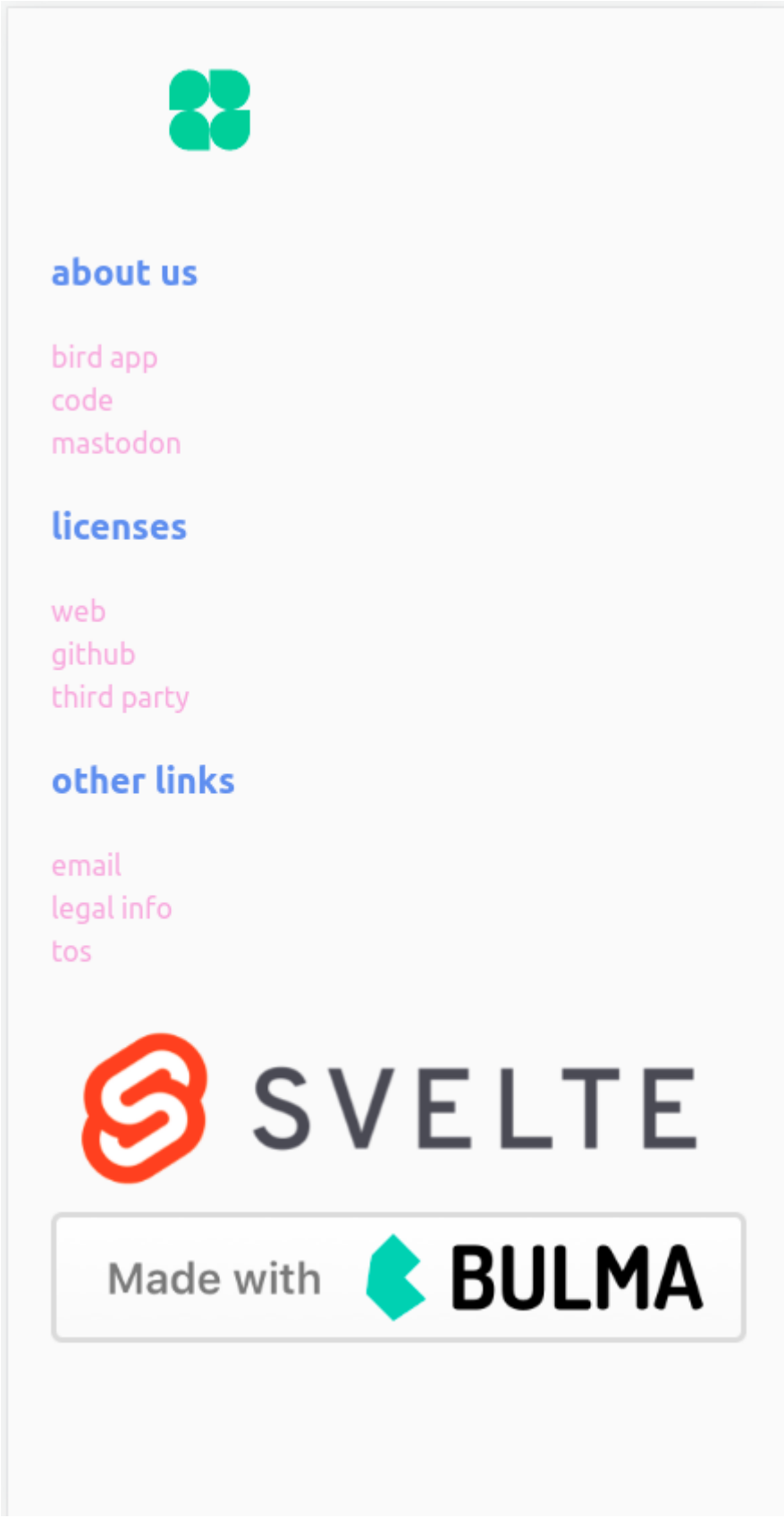

Figura 6.10: Pantalla de un kiosko con tamaño móvil – vista del footer

Podemos observar ahora en las figuras Figura 6.6, Figura 6.7, Figura 6.8, Figura 6.9 y Figura 6.10 cómo los tamaños de todos los componentes se han adaptado a la nueva pantalla.

Mapa

Para definir el mapa a partir de las coordenadas que *Django* pueda devolver, se ha elegido *Leaflet*[22] como librería.

Leaflet convierte unas coordenadas en un mapa de manera muy simple:

```
<script lang='ts'>
   import { onMount } from 'svelte';
   import { browser } from '$app/env';
   import { Location } from '../../api/models/location';

   export let location: Location;
   export let markerText: '';
   onMount(async () => {
      if (browser) {
         const leaflet = await import('leaflet');

         const map = leaflet.map('map').setView([location.latitude,
             location.longitude], 13);

         leaflet.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png').addTo(map);

         leaflet.marker([location.latitude, location.longitude]).addTo(map)
            .bindPopup(markerText)
            .openPopup();
      }
   });
</script>

<main>
   <div id='map'></div>
</main>

<style>
    @import 'https://unpkg.com/leaflet@1.7.1/dist/leaflet.css';

    main #map {
        height: 400px;
    }
</style>
```

Simplemente hay que importar la librería, crear un div al que se le asigna un id y un tamaño, asignarle ese id al objeto *leaflet* y añadirle las coordenadas necesarios. Además de eso, también se le puede pasar texto personalizado para el marcador que indica el punto establecido por las coordenadas.

Además de mostrar un punto en un mapa, *Leaflet* ofrece un gran nivel interactividad, pudiendo moverse alrededor utilizando el ratón, además de aumentar o encoger la vista que se tiene del mapa.

[22] https://leafletjs.com/

6.4 Seguridad

En este apartado se discutirán los aspectos más importantes de la seguridad y los porqués de algunas decisiones tomadas en pos de este aspecto.

6.4.1. Seguridad de la plataforma base

Django ofrece una serie de medidas de seguridad por defecto. Entre ellas:

- Medidas implementadas por defecto:
  - *Host header validation*, una validación simple donde el servidor solo admite peticiones de una lista de hosts especificados en los ajustes del sistema.
  - Cross-site request forgery (CSRF) es una medida de seguridad que protege contra ataques donde un posible agresor pueda falsificar peticiones al servidor utilizando las credenciales de sesión del sitio web.

    Por ejemplo: un usuario victima se autentica en la página web de un banco, y en ese proceso, guarda ciertos credenciales que autorizan una sesión, donde el usuario puede realizar peticiones de manera continua con dicha sesión. Si un agresor consigue que la victime haga click en un enlace que realice operaciones no deseadas por la víctima en el servidor, este puede utilizar las credenciales de sesión para autenticar esa petición.
  - Inyección Structured Query Language (SQL). Este tipo de medida solo funciona mientras los programadores usen las funciones de más alto nivel para manipular la base de datos, puesto que estas nunca piden formar una cadena válida SQL, ya que la manipulación de datos
  - solo se hace utilizando parametrización de funciones del lenguaje ofrecidas por el ORM.

    Si por ejemplo, para buscar el kiosko con *SomeKiosk* como valor de su su atributo *handler* este tiene que introducir que introducir la URL

    `/kiosks/?handler=SomeKiosk`

    podría, maliciosamente, en vez de ese valor, poner

    `/kiosks/?handler='; DROP TABLE kiosks'`

    y a priori eliminaría la tabla *kiosks*. Sin embargo, cuando esa petición llegue al servidor, este no lanzará una consulta tipo

    `SELECT * FROM kiosks where kiosks.name=; DROP TABLE kiosks`

    sino que pasará por una función del ORM parecida a

    `Kiosk.objects.get(name="; DROP TABLE kiosks")`

    sanitizando así la entrada: para cuando esa operación del ORM se traduzca a una consulta SQL, ese valor se interpretará literalmente como una cadena de caracteres.

- Medidas a implementar:
  - La subida de archivos multimedia debería hacerse desde un dominio de segundo nivel en comparación con el dominio principal del servidor, prohibiendo así cualquier exploit bloqueado por el concepto de *same-origin policy*, donde cualquier script que se ejecute desde un dominio diferente al principal no es aceptado

- Medidas que mitiguen ataques de denegación de servicios, sobre todo a la hora de, como en el caso anterior, manejar archivos multimedia que suelen tener un peso considerable. Una medida básica sería limitar el número de subidas en un rango de tiempo desde un mismo origen, así como limitar el tamaño máximo por archivo o petición

6.4.2. Autenticación

Aunque el backend, al ser una aplicación *Django* tiene que ofrecer la autenticación basada en sesiones porque la aplicación de administración está habilitada, el método preferido de autenticar el frontend es el flujo `code` con Proof Key for Code Exchange (PKCE) simplificado, donde:

- El frontend genera un secreto aleatorio que se guarda en una variable llamada `code_challenge`.
- El usuario introduce sus credenciales y el frontend los envía junto a un hash -como mínimo un SHA-512- de la variable `code_challenge` al servidor para recibir un código llamado `authorization_code`
- El backend envia el `authorization_code` junto al valor de `code_challenge` y a su `client_id`, un token que identifica a este cliente, para recibir otro token con el que poder realizar operaciones

Autenticación del chat

La autenticación del chat se implementa enviándole al servidor el `access_token` del usuario autenticado en cuanto la conexión *WebSocket* se establece. Este protocolo, al contrario que HTTP, sí tiene estado: esto significa que una comunicación *WebSocket* solo funciona si previamente se ha establecido un canal por el que enviar mensajes. Esto significa que solo se tienen que enviar credenciales una vez, pues todas las siguientes llamadas del mismo canal estarán autenticadas.

6.4.3. Manejo de las URL

Un aspecto importante de la seguridad implementada es cómo se manejan los identificadores de los recursos. Se ha decidido no proveer a ningún cliente con identificadores básicos -campo id- de cualquier recurso, ya que estos son secuenciales en la base de datos se podría inferir información prohibida. Además, es mucho más fácil para un usuario recordar una combinación gramaticalmente lógica de palabras que un número, por lo que los recursos que Agropop ofrece vienen identificados, a nivel público, por un *handler*. Todos los recursos creados pueden tener este atributo asignado automáticamente por el sistema, aunque el usuario, si es dueño, puede cambiarlo.

De manera más gráfica, tradicionalmente se accedería a una tienda utilizando el id de su base de datos: `/shops/450/` pero con la nueva solución, la identificación por URL de la misma se haría utilizando un texto compuesto de varios adjetivos y un sustantivo `/shops/SomeFantasticShop/` o cualquier cadena que el usuario desee, por ejemplo: `/shops/MiTiendaDeFrutasDeVerano/`.

6.5 Despliegue

En este apartado hablaremos de cómo desplegar todos los componentes de Agropop, incluyendo instrucciones para herramientas externas de cierta popularidad.

6.5.1. Backend

La implantación del backend implica el despliegue de ciertos componentes externos a Agropop y el ajuste de este para que se comunique de manera correcta con los componentes mencionados. Por otro lado, este despliegue se realiza sobre un sistema GNU/Linux.

Los componentes necesarios son: *PosgreSQL* como base de datos, *Redis* como base de datos en memoria y un *proxy inverso* que en este caso es *Nginx* pero sirve cualquiera que cumpla las características que se explicarán a continuación.

De estos servicios se necesita que previamente se hayan configurado una serie de parámetros. Para *PosgresSQL* y *Redis*, es necesario crear una base de datos y un usuario asignado a dicha base de datos con todos los permisos posibles. Para el *proxy inverso* solo es necesario que se pase la cabecera `Upgrade`. En `Nginx` sería algo parecido a:

```
location / {
        proxy_pass http://agropop;
        proxy_http_version 1.1;
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_set_header Upgrade $http_upgrade; # esta es la cabecera
            necesaria
        proxy_set_header Connection $connection_upgrade;
}
```

Esta cabecera permite actualizar la conexión de HTTP a *WebSocket* cuando se necesario. De lo contrario, el chat no funcionaría.

Una vez tengamos los componentes externos preparados, es hora de preparar el propio Agropop. Primero será necesario crear una carpeta donde se guarden los ficheros multimedia. Luego será necesario crear un fichero llamado `.env` donde se guardaran variables de entorno que especifiquen los ajustes de Agropop. En dicho fichero se pondrán, a priori los ajustes que no tengan una opción por defecto, aunque estos también son modificables. Un ejemplo:

```
# PROJECT META VARS
PROJECT_NAME=agropop
PROJECT_DESCRIPTION="Backyard-grown produce trading"
PROJECT_FQDN=agro.pop
PROJECT_SECRET_KEY="quote it if using special characters"
SITE_ID=1
TIME_ZONE=UTC

#  'True' or 'False', needs to be python-compliant since
# it'll be cast to a boolean value
DEBUG=True

# comma-separated
ALLOWED_HOSTS=localhost,127.0.0.1

# DATABASE
DATABASE_HOST=localhost
DATABASE_NAME=agropop
DATABASE_USER=agropop
DATABASE_PASSWORD=agropop
DATABASE_PORT=5432

# STATIC
STATIC_URL=/static/
MEDIA_URL=/media/
MEDIA_ROOT=/home/agropop/media/
```

Estas variables obligan a Agropop a, entre otras cosas, aceptar peticiones solo de `localhost` o `127.0.0.1`, conectarse a una base de datos llamada `agropop`, con usuario y contraseña con el mismo nombre. Además define una carpeta para guardar los ficheros multimedia en la variable `MEDIA_ROOT`

Podemos así, lanzar el servicio web Agropop lanzando el comando `python manage.py runserver`, mientras nos encontremos en la carpeta raíz del proyecto, y ver que funciona en la URL `http://localhost:8000`. En la URL `http://127.0.0.1:8000/api/schema/swagger-ui/` tenemos documentación sobre todos los endpoints disponibles, tal como en la figuras Figura 6.11 Figura 6.12

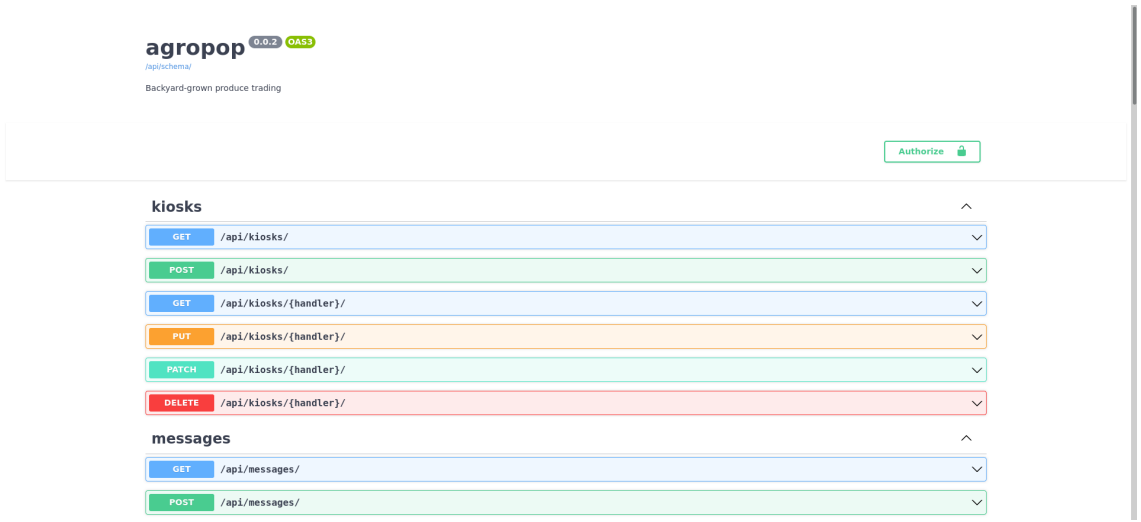

Figura 6.11: Vista de la documentación Swagger del backend Agropop

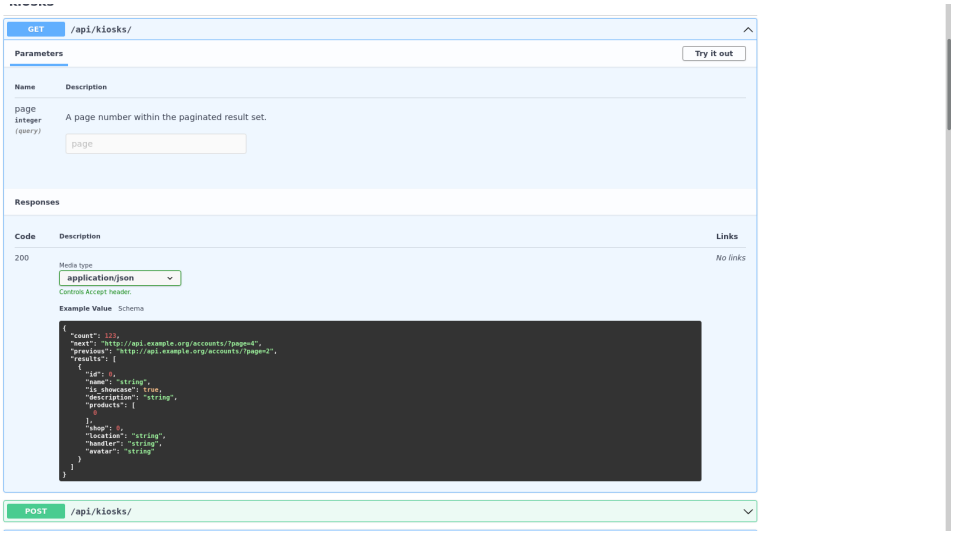

Figura 6.12: Vista de la documentación Swagger del backend Agropop

En un entorno de producción, Agropop debería ser convertido en un `demonio`, esto es, debería estar corriendo en segundo plano de manera ininterrumpida. En casi todas las distribuciones modernas GNU/Linux esto se consigue escribiendo un *unitfile* de *systemd*, el programa encargado de gestionar servicios.

Un *unitfile* muy básico para Agropop sería:

```
#agropop.service

[Unit]
Description=Agropop server

After=network.target postgresql.service

[Service]
User=agropop
Group=agropo
EnvironmentFile=/home/agropop/.env
ExecStart=/usr/bin/python /home/agropop/agropop/manage.py runserver
WorkingDirectory=/home/agropop/agropop

[Install]
WantedBy=multi-user.target
```

Entre varias cosas, se define el espacio de trabajo -*WorkingDirectory*-, el cual es la carpeta raíz de Agropop, el comando completo para lanzarlo -*ExecStart*- y el fichero que contiene las variables de entorno anteriormente definidas -*EnvironmentFile*.

Solo queda lanzar este servicio mediante dos comandos:

```
systemctl daemon-reload
```

```
systemctl start agropop.service
```

6.5.2. Frontend

En contraste con el backend, el frontend puede ser un como más complejo a la hora de desplegar, aunque a la vez resulta más simple porque tiene menos pasos. Es más complejo porque SvelteKit es un compilador, tal como hemos mencionado anteriormente, lo que

signifca que, obviando el entorno de desarrollo, quien lanza el código tiene que ser otra herramienta. Para esto se ha creado el concepto de adaptador donde, se define qué tipo de compilación se quiere. Destacamos dos tipos: adaptador estático o adaptador para *NodeJS*.

El primer adaptador compila el proyecto a ficheros estáticos, HTML, CSS y *JavaScript*. Estos pueden ser servidos luego por, por ejemplo, el anteriormente mencionado *Nginx* como una página web estática.

El segundo adaptador compila a un fichero `index.js` para que se lanzado por *NodeJS*. El proyecto tiene este segundo puesto como adaptador por defecto, por lo que solo hay que compilar con el comando `yarn build` y ejecutar el nuevo fichero cread con `node build/index.js`

Antes de lanzar el programa, y al igual que en el apartado anterior, también vamos a necesitar un fichero *.env* para definir ciertos parámetros de comunicación con el backend:

```
VITE_API_URL = "http://localhost:8000/api"
VITE_AUTH_URL = "http://localhost:8000/o/token/"
VITE_SITE_NAME = "agropop"
VITE_CLIENT_ID = "KKuCp6IWh6KRK7RBES6Z5S7wjr7KIYbtrqNFIjKj"
VITE_CLIENT_SECRET =
    "2oyf4MsuhyXWlEFCv6RnQ8nBKWXKOyFBCcGfhPJl5cj5Tn7tjD3drqpWQcndxLvw0FH9qHh4i3ETPv2e6tq5S3JmKgws8
```

Las dos primeras variables definen endpoints del backend, siendo el primero el que define el API del mismo, mientras que el segundo define a qué URL pedir un token de acceso. Los dos últimos son credenciales que identifican a este frontend y son necesarios para que el servidor devuelva un token de acceso cuando un usuario incia sesión.

De nuevo siguiendo los mismos pasos que en el apartado anterior, este programa debería ejecutarse como un servicio, teniendo un *unitfile* parecido.

```
#agropop-web.service

[Unit]
Description=Agropop web server

After=network.target agropop.service

[Service]
User=agropop
Group=agropo
EnvironmentFile=/home/agropop/web/.env
ExecStart=/usr/bin/node /home/agropop/web/build/index.js
WorkingDirectory=/home/agropop/web

[Install]
WantedBy=multi-user.target
```

En cuanto al proxy inverso, los mismos ajustes que en backend se aplican para que el chat funcione.

CAPÍTULO 7
Pruebas

En este capítulo vamos a explicar cómo se ha a integrado el proceso de probado en el proyecto, haciendo mención a la metodología aplicada y a técnicas más novedosas como la integración continua.

7.1 TDD

Desarrollar pensando primero en los tests tiene ciertas implicaciones, tanto a nivel de código como a nivel organizativo. Esto es, el proceso de creación de las pruebas empieza cuando una tarea es definida, integrando TDD con el concepto de INVEST anteriormente visto, una historia de usuario es definida con sus pruebas de aceptación, las cuales definen acaban definiendo los comportamientos que el programa tendrá una vez estas pruebas pasen comienza el proceso propio de TDD. A partir de este momento, comienza el proceso :

1. Una prueba de aceptación se transforma en un test en el lenguaje de programación del sistema
2. Se comprueba que la prueba efectivamente falla porque no se ha realizado ningún tipo de programación adicional
3. Se realiza dicha programación
4. Si comprueba que el test no falla y se reprograma hasta que este pase
5. Si el test pasa, es imperativo ejecutar todos los demás para comprobar que no hay efectos laterales
6. Si todos los tests pasan, se refactoriza el código, quitando *impurezas* y otros defectos
7. Se vuelven a pasar todos los tests, comprobándose que finalmente el nuevo código añadido es correcto

7.2 Integración y despliegue continuos

A lo largo de los últimos años, sumado a técnicas como la explicada en el apartado anterior, han ido surgido ciertas metodologías llamadas Continuous Integration/ Continuous Delivery (CI/CD) que permiten automatizar el proceso de testeo y despliegue si

este ha sido correcto. La base de este proceso consiste en ejecutar una secuencia de comandos una vez el código es subido a la plataforma de control de versiones, replicando lo que el programador debería hacer antes de subir dicho código.

Poniendo como ejemplo el backend de Agropop, una herramienta llamada `pre-commit` es utilizada,la ejecuta una serie de comprobaciones, tanto estáticas (linting) como dinámicas (testeo). Si esas comprobaciones no pasan satisfactoriamente, el programador no puede realizar un commit de sus cambios. Este mismo proceso es luego utilizado por la plataforma CI/CD para establecer un entorno desde cero, instalando todas las dependencias necesarias, y ejecutando los mismos pasos que la herramienta `pre-commit` e incluso pasos adicionales.

Se puede observar en la Figura 7.1 una serie de pasos en un grafo. Primero el proyecto es clonado por la plataforma de integración continua, se crea un entorno en separado para ejecutar la base de datos, se instalan las dependencias, se realiza el la comprobación estática y luego se ejecutan los tests.

Esta operación puede complicarse infinitamente dependiendo de las necesidades del equipo. En general existen unos pasos más adicionales que no aparecen en la captura, donde, dependiendo de la rama del código que la plataforma ejecute, pueda desplegarlo para tener acceso directo a una ejecución del servicio. Todo esto resulta interesante, sobre todo, en sistemas de control de versiones que permitan un paso previo a subir código de una rama de desarrollo a una rama no efímera, donde existe un sistema de aprobación de esa fusión de ramas por parte de los dueños del repositorio. En plataformas como *Github* este proceso se conoce como *pull request* mientras que otras como *Gitlab* lo llaman *merge request*.

Teniendo todo esto en mente, se pueden elegir varios grados de automatización en función de la madurez del sistema de integración continua presente. Pueden existir, por ejemplo, casos donde, en entornos no cruciales, los *merge/pull requests* sean aceptados automáticamente al pasar las pruebas.

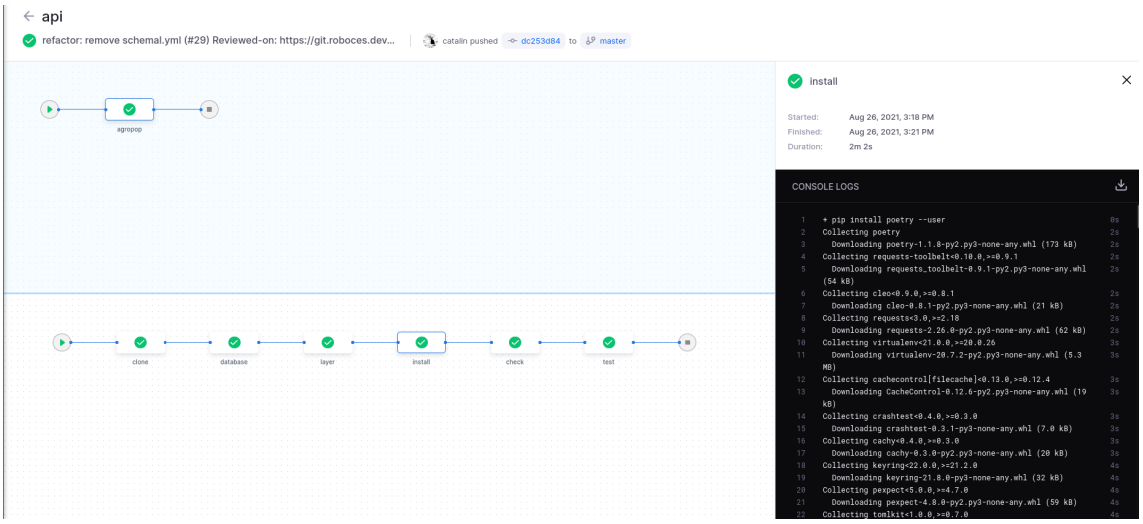

Figura 7.1: Vista del paso de instalación de la operación CI/CD de Agropop

7.3 Métricas de rendimiento

Antes de analizar las métricas de rendimiento, es imperativo mencionar que estas se han hecho sobre una máquina óptima para los tipos de carga probados.

Mencionar que en la base de datos existen 50 tiendas, 50 usuarios, 2550 kioskos y 37500 productos.

Todas las métricas son tomadas por el propio servidor utilizando la librería *django-silk*[1]

7.3.1. Métricas con carga continua

Se han realizado 2400 peticiones al servidor de manera automática a todos los *endpoints* disponibles. Este proceso ha durado un total de 272.2 segundos. No mostraremos el desglose por cada petición pero sí que podemos extraer datos interesantes:

- El tiempo medio por petición es de 27ms
- El número medio de consultas a la base de datos es de 2.33 por petición
- El tiempo medio de acceso y vuelta de la base de datos por cada petición es de 4ms
- Las operaciones más costosas han sido las de listado de datos:
  - `/api/kiosks/` con un tiempo total de 112ms, de los cuales existen 12 consultas a la base de datos en un tiempo total de 73ms
  - `/api/users/` con un tiempo total de 73ms, de los cuales existen dos consultas a la base de datos en un tiempo total de 1ms
  - `/api/shops/` con un tiempo total de 68ms, de los cuales existen dos consultas a la base de datos en un tiempo total de 1ms
  - `/api/users/user_1` con un tiempo total de 59ms, de los cuales existe una consulta a la base de datos en un tiempo total de 1ms
  - `/api/kiosk/shop_0_kiosk_0` con un tiempo total de 58ms, de los cuales existen dos consultas a la base de datos en un tiempo total de 8ms

7.3.2. Métricas en solitario

En este apartado analizaremos las tres consultas más costosas del anterior apartado, repitiéndolas cada una cinco veces, por separado, estableciendo una media y una mediana:

Endpoint	1		2		3		4		5		Media		Mediana	
/api/kiosks	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt
	55	20	52	15	66	23	58	18	54	14	57	18	55	18
/api/users	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt
	19	1	19	1	20	1	19	1	21	1	19.6	1	19	1
/api/shops	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt	Ot	DBt
	27	2	31	2	26	2	25	2	30	2	27.8	2	27	2

Tabla 7.1: Tabla con resultados del análisis de métricas

La cabecera `Ot` significa *Overall time* o tiempo total de una petición, mientras que `DBt` hace referencia a la fracción de tiempo que ha tardado la base de datos.

[1] https://github.com/jazzband/django-silk

7.3.3. Conclusiones

Podemos concluir que el endpoint más costoso es `/api/kiosks/`, el cual en el primer análisis ha ofrecido un valor a priori, anómalo, bastante alto, de 112ms, mientras que el valor normal suele ser de 57ms como media y 55 como mediana.

Como este fenómeno también ocurre con los otros dos endpoints más costosos, donde el coste cuando la carga es relativamente intensa aumenta considerablemente, podemos inferir que el servidor tiene un cuello de botella con en el número de peticiones que puede admitir. Una potencial solución a esto, si la conclusión es correcta, es poner un sistema de colas que gestione las operaciones según los recursos en un momento dado del sistema. Este es un aspecto ya pleando que se ve explicado en un apartado posterior.

Por otro lado, hemos descubierto que uno de los endpoints hace una cantidad inusual de peticiones a la base de datos, `/api/kiosks/` con 12 peticiones. Este número es demasiado alto para una operación tan sencilla, teniendo en cuenta que las peticiones homónimas como `/api/shops` o `/api/products` tienen solo dos consultas a la base de datos.

Si ahondamos en este último problema mirando exactamente qué consultas se hacen a la base de datos, tenemos que el endpoint de los kioskos tiene una primera consulta simple `SELECT ... FROM ... LIMIT 10;` seguida de 11 consultas una tabla intermedia entre productos y kioskos, donde cada una presenta un `INNER JOIN`. Esto puede indicar que la definición de los modelos a nivel de ORM puede no ser correcta y hay que planear una refactorización.

CAPÍTULO 8
Conclusiones

Al principio de esta memoria se han definido un número de objetivos a cumplir. Creemos que a lo largo de esta, estos objetivos han quedado demostrados como cumplidos.

Agropop provee una plataforma donde un usuario puede hacer acciones de vendedor o comprador, siendo estas la posibilidad de crear una tienda, kioskos asignados a la misma para categorizar una serie de productos que el vendedor ofrece. Por su lado, el comporador puede interactuar con una tienda, kioskos y productos, pudiendo establecer una sala de chat para cada producto y así iniciar un proceso de compra del mismo. Asimismo, el comprador puede interactuar con un mapa en dos dimensiones por cada kiosko para establecer dónde este se encuentra físicamente.

Introduciendo el siguiente apartado, estos objetivos se han cumplido aplicando ciertas metodologías y arquitecturas que se han visto a lo largo de los estudios cursados

8.1 Relación con los estudios cursados

Casi todos los aspectos tratados aquí han sido posibles gracias a los estudios cursados. Tanto en el aspecto metodológico y abstracto como en el aspecto más técnico.

El concepto de metodología ágil y, sobre todo, la aplicación de esta mediante Scrum se ha ido viendo de manera progresiva a lo largo de la carrera, habiéndose realizado varios proyectos en grupo aplicándola. Este hecho sirvió, evidentemente, como base y, a lo largo del trabajo, se fue adaptando según las necesidades del mismo, añadiendo paralelamente otras metodologías que cubriesen aspectos no contemplados, como TDD, o ahondado de manera más real y específica en el propio proceso.

Otro aspecto visto en los estudios cursados y que ha tenido mucha incidencia en el proyecto han sido los análisis de alto nivel del problema, en específico, toda la serie de diagramas creados para entender y especificar el problema. Todos los diagramas de casos de uso, de clases, de componentes, etc. han podido realizarse porque existía un bagaje previo adquirido a lo largo de los cursos.

También resulta imperativo mencionar otros conceptos que no se han explicado en este trabajo pero sí se han estudiado, sobre todo, en las asignaturas de la rama de Ingeniería del software y se han aplicado en este proyecto. Concretamente, es el caso de los patrones de diseño y los principios de desarrollo como Single-responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion (SOLID) o Don't Repeat Yourself (DRY). En el caso del bagaje de patrones de diseño adquirido, no solo han servido como base para desarrollar ciertas partes del proyecto en sí, sino también para interpretar

mejor patrones no vistos, como es el caso de la equivalencia que una de las plataformas hace entre el patrón Model View Controller (MVC) y MVT explicado con anterioridad.

8.2 Trabajos futuros

En este capítulo mencionaremos en más detalles aspectos que Agropop desea adquirir en un futuro. Algunos de ellos ya se han mencionado con anterioridad a lo largo de este trabajo.

8.3 Federación y autenticación externa

En el contexto informático, la federación tiene una serie de significados dependendiendo del problema a abordar. En este caso, y reincidiendo en lo expuesto cuando se mencionó la plataforma *Mastodon*, la federación que Agropop busca es una donde distintas implantaciones de Agropop puedan comunicarse entre sí.

Por ejemplo, en una ciudad dada, cada vecindario que lo necesitase implanta su propio Agropop. Sin embargo, mediante federación, estos usuarios podrían realizar las operaciones de su nodo en los demás si lo necesitasen, así como publicar sus propias tiendas y kioskos para que sean visibles para toda la red de sistemas Agropop.

Un aspecto tangencial al anterior es la posibilidad de poder autenticar los usuarios utilizando plataformas que también utilicen el modelo *Oauth2*, como Google, eximiendo a los usuarios de la necesidad de tener sus credenciales en Agropop.

8.4 Plataformas de pagos y operaciones recurrentes

Un aspecto que Agropop aún no contempla son las plataformas de pago, lo que significa que una vez un comprador accede al trato con un vendedor, estos tienen que buscar una manera externa de realizar la eventual operación económica. Idealmente se buscaría integrar este aspecto Agropop con la ayuda de plataformas de pago externas como Strip[1] o PayPal[2]

Implementar estas plataformas ayudaría con el despliegue de otro aspecto: las operaciones recurrentes. Poder vender o repartir productos a un consumidor utilizando un sistema de suscripciones.

8.5 Tiendas customizables

Un aspecto muy importante a la hora de construir plataformas de comercio electrónico es dar la posibilidad de customizar la tienda -y en este caso, kioskos- a los vendedores. En Agropop planeamos tener esta característica prestando ideas de otras plataformas que ya hacen algo parecido de una forma muy simple, mediante el modelo *drag and drop*, donde se provee al dueño de una tienda con una serie de componentes arrastables y customizables. Se pueden ver ejemplos de esto en plataformas como Odoo[3].

[1] https://stripe.com
[2] https://paypal.com
[3] https://youtu.be/Sl0a8-16-Hk

8.6 Aspectos más técnicos

8.6.1. Añadir un sistema de colas

Resultará interesante añadir un sistema externo de colas para gestionar las operaciones que el servidor recibe, pues *Django* funciona de manera síncrona: una operación llega, es procesada y finaliza, liberando los recursos que esa operación tomó solo cuando esta ha finalizado. Idealmente este problema se vería solucionado si *Django* utilizara el mismo esquema asíncrono que utiliza la librería *channels*, empleada para desarrollar el chat. Sin embargo, otra alternativa es integrar un sistema de colas como *Celery*[4], al cual todas las peticiones se le pasan, las encola, y el sistema las procesa según los recursos se lo permiten.

8.6.2. Búsqueda full-text

Como se ha mencionado, la búsqueda full-text ahora mismo se realiza mediante la librería *Watson*, la cual no es muy eficiente en cuanto el sistema escala a cierto número de usuario. Teniendo un volumen de usuarios o peticiones más grando, una mejor solución es *Elasticsearch*[5].

8.6.3. Autenticación del chat

Como se ha visto explicado anteriormente, la autenticación del chat es muy personalizada, sin seguir ningún tipo de estándar. Idealmente esto debería resolverse haciendo que la comunicación *WebSocket* heredara las cabeceras de autenticación de la comunicación HTTP normal que la invoca.

[4] https://docs.celeryproject.org/en/stable/
[5] https://www.elastic.co/elasticsearch/

Glosario

backend en un contexto computacional, *backend* hace referencia al sistema con el que el usuario no puede interaccionar directamente, el cual suele ser responsable de la lógica de guardado y tratamiento de datos.

backlog en Scrum, backlog hace referencia a la lista de historias de usuario no asignadas a ningún sprint.

commit commit es un concepto que muchos sistemas de control de versiones comparten, aunque no para todos significa lo mismo. Por ejemplo, para Git, commit implica guardar los cambios realizados en el árbol de trabajo local actual, asignando una estampa con fecha y autor a tal cambio.

compilador en un contexto computacional, un compilador hace referencia a un programa que transforma un código fuente de un programa a otro, ya sea este entendible por humanos o no (binario).

código libre el código libre hace referencia a código que respeta la liberdad de los usuarios y su comindad, teniendo que cumplir con cuatro definiciones de libertad definidas por la Free Software Foundation en 1986.

decorador algunos lenguajes como `Python` o `Typescript` implementan una técnica para modificar o añadir funcionalidad a funciones existentes sin modificar el código de la propia función pero añadiendo un elemento encima de la definición de la misma, generalmente con el formato `@decorator(variable_o_funcion)`.

deserializar es el proceso contrario al de serialización, donde se convierte de un formato estándar a un objeto que un programa dado pueda entender nativamente.

div elemento HTML para definir secciones o contenedores de otros elementos.

exploit en inglès, un *exploit* es un una operación informática usada para aprovechar una vulnerabilidad para causar comportamientos no previstos en un componente electrónico, ya sea a nivel de hardware o software.

framework en un contexto computacional, *framework* hace referencia a un sistema de abstración redistribuible y reutilizable en otros entornos, dictando de manera dogmática comportamientos que un programa y equipo de desarrollo debe seguir.

frontend en un contexto computacional, *frontend* hace referencia al sistema con el que el usuario interactúa directamente.

generalización en el contexto de los diagramas UML, en especial los diagramas de casos de uso y clases, la generalización implica una relación entre dos casos o clases donde el caso o clase hijo hereda comportamientos y atributos del padre, siendo así el hijo más especializado y el padre más abstracto.

hash un *hash* es el resultado de una operación de mapeo de de una cantidad arbitraria de datos a una de tamaño fijo e, idealmente irrepetible. Algunos ejemplos de estas tecnologías son: SHA-1, SHA-256, MD5.

id diminutivo de identificador, secuencia de caracteres, generalmente números, que suele identificar un recurso en una base de datos o cualquier agrupación de elementos.

linting un *linter* es una herramienta de análisis estático de código, donde se busca encontrar bugs y errores de formato.

monolítico en un contexto computacional, una arquitectura monolítica es aquella que abarca todas las soluciones que el problema presenta. Es la contra posición a una arquitectura *bazar*, donde un programa es dividido entre varios múltiples componentes atómicos que se comunican entre sí para generar una estructura superior.

rama en *Python* y otros lenguajes, diccionario hace referencia a lo que más comúnmente se conocen como tablas *hash*, estructuras de datos tipo clave-valor donde la clave es un identificativo único de un solo valor.

rama en sistemas de control de versiones, el concepot de rama suele hacer referencia a desarrollos paralelos al principal, siendo una analogía de las ramas de un árbol.

script en un contexto computacional, un script hace referencia a un programa que ejecuta todas sus funciones en orden secuencial, igual que el guión -*script*- de una obra de teatro.

serializar en un contexto computacional, serializar significa convertir objetos máquina, como por ejemplo los objetos de un lenguaje como Java, a un formato que se pueda transmitir por red para ser transformado en un objeto por parte del sistema o lenguaje que recibe esta serialización. En general, serializar implica convertir un objeto a un lenguaje de texto plano o estándar como JSON.

sprint en el contexto de las metodologías ágiles, en especial Scrum, sprint hace referencia a un periodo de tiempo, generalmente corto, de entre una y cuatro semanas, donde se realiza un trabajo previamente planificado en tareas atómicas.

token en un contexto computacional, *token* hace referencia a un elemento criptográficamente creado en sustitución de un recurso, identificándolo sin revelarlo en público.

Acrónimos

API Application Programming Interface.

CI/CD Continuous Integration/ Continuous Delivery.

CRUD Create, read, update, delete.

CSRF Cross-site request forgery.

CSS Cascading Style Sheets.

DOM Document Object Model.

DRF Django REST Framework.

DRY Don't Repeat Yourself.

GDPR General Data Protection Regulation.

GIS Geographic Information System.

GPL GNU Public License.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

INVEST Independent, Negotiable, Valuable, Estimable, Small, Testable.

JSON JavaScript Object Notation.

MIME Multipurpose Internet Mail Extensions.

MVC Model View Controller.

MVP Minimum viable product.

MVT Model View Template.

OAS OpenAPI Specification.

ORM Object-relational mapping.

PKCE Proof Key for Code Exchange.

REST Representational state stransfer.

SEO Search Engine Optimization.

SOLID Single-responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion.

SQL Structured Query Language.

SSR Server-side rendering.

TBD Trunk Based Development.

TCP Transmission Control Protocol.

TDD Test-driven development.

UML Unified Modeling language.

URL Uniform Resource Locator.

WIP Work In Progress.

Bibliografía

[1] Christopher Isett, Stephen Miller. *The Social History of Agriculture: From the Origins to the Current Crisis*. Rowman & Littlefield Publishers, 2016

[2] Cockburn, Alistair. *Writing effective use cases*. Addison-Wesley Professional, 2000;2007

[3] Manuel González de Molina, David Soto Fernández, Gloria Guzmán Casado, Juan Infante-Amate, Eduardo Aguilera Fernández, Jaime Vila Traver, Roberto García Ruiz *The Social Metabolism of Spanish Agriculture, 1900–2008: The Mediterranean Way Towards Industrialization*. SpringerOpen, 2019

[4] Especificación OpenAPI 3.0. Consultada a https://swagger.io/specification/.

[5] The Practical Test Pyramid. Consultada a https://martinfowler.com/articles/practical-test-pyramid.html

[6] ScopeLimbering. Consultada a https://martinfowler.com/bliki/ScopeLimbering.html

[7] StandardStoryPoints. Consultada a https://martinfowler.com/bliki/StandardStoryPoints.html

[8] GeoDjango Model API Consultada a https://docs.djangoproject.com/en/3.2/ref/contrib/gis/model-api/

[9] Make a Location-Based Web App With Django and GeoDjango Consultada a https://realpython.com/location-based-app-with-geodjango-tutorial/#displaying-nearby-shops

[10] Implement a Chat Server Consultada a https://channels.readthedocs.io/en/stable/tutorial/part_2.html

[11] Media types. Consultada a https://www.iana.org/assignments/media-types/media-types.xhtml

[12] Frequently Asked Questions about the GNU Licenses Consultada a https://www.gnu.org/licenses/gpl-faq.html

[13] Virtual DOM is pure overhead Consultada a https://svelte.dev/blog/virtual-dom-is-pure-overhead

[14] UserStory Consultada a https://martinfowler.com/bliki/UserStory.html

[15] Introduction to the DOM https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

[16] Window https://developer.mozilla.org/en-US/docs/Web/API/Window

[17] Best Practices - OAuth for Single Page Applications https://curity.io/resources/learn/spa-best-practices/

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**	X			
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**	X			
ODS 9. **Industria, innovación e infraestructuras.**		X		
ODS 10. **Reducción de las desigualdades.**	X			
ODS 11. **Ciudades y comunidades sostenibles.**	X			
ODS 12. **Producción y consumo responsables.**	X			
ODS 13. **Acción por el clima.**		X		
ODS 14. **Vida submarina.**			X	
ODS 15. **Vida de ecosistemas terrestres.**			X	
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Aunque creemos que el proyecto Agropop guarda un grado mínimo de relación con hasta 8 ODS, a continuación contextualizamos los más interesantes.

Entre otras metas para el año 2030 que el objetivo *Hambre cero* propone, están: "duplicar la productividad agrícola e ingresos de productores de alimentos en pequeña escala", y "asegurar la sostenibilidad de lo sistemas de producción de alimentos y aplicar prácticas agrícolas resilientes que aumenten la productividad y la producción", las cuales encajan perfectamente con la propia razón de ser del proyecto, que busca eliminar la necesidad de intermediarios en los mercados agrícolas mediante una plataforma de negocio auto-implantable, permitiendo a los propios productores gestionar la vida mercantil de sus productos y a los compradores acceder a precios más justos o incluso utilizar sistemas no basados en moneda, como podría ser una gestión no mercantil de la industria agrícola de una zona dada.

Los hechos anteriores pueden enlazar con el objetivo número nueve, *Industria, innovación e infraestructuras*, ya que Agropop permitiría, en cierta medida, un principio de "industrialización inclusiva y sostenible"; los dos mayores problemas que la industria agrícola afronta son los precios abusivos impuestos por la larga cadena de intermediarios y la poca competencia que el pequeño comercio puede hacer contra pilares de la industria a nivel de servicio prestado y precios – en cualquier momento estos pilares pueden dictar las reglas del juego obligando a los competidores más pequeños y particulares a poner, por ejemplo, precios que suponen pérdidas. El proyecto desarrollado puede ayudar a combatir estos problemas ofreciendo una plataforma donde los pequeños comerciantes y particulares pueden establecer un comercio online sin necesidad de grandes presupuestos, así como formar una red de diferentes plataformas online donde compartir clientes.

Tomando lo anterior como referencia, creemos que Agropop también puede ayudar en una medida por lo menos mínima a cumplir ciertas metas del objetivo *Ciudades y comunidades sostenibles* donde, gracias a la ayuda al pequeño comercio en un sector tan importante como la agricultura, se apoyan "los vínculos económicos, sociales y ambientales positivos entre las zonas urbanas, periurbanas y rurales fortaleciendo la planificación del desarrollo nacional y regional" y se facilita "el desarrollo de infraestructuras sostenibles y resilientes en los países en desarrollo".

Por otro lado, aunque muy cercanos a los puntos anteriores, el objetivo *Producción y consumo responsable* también es uno que pertenece de manera involuntaria a la razón de ser de Agropop. Según los datos sobre el apartado Comida ofrecidos, "se calcula que un tercio de todos los alimentos producidos, [...], termina pudriéndose en los contenedores de los consumidores y minoristas, o se estropea debido a las malas prácticas del transporte y la cosecha". El proyecto Agropop ayudaría a evitar en cierta parte estos problemas ya que consideramos que el consumidor es más consciente de su consumo cuando este es en un comercio más comunitario. Por otro lado, y el punto más importante: si la agricultura consigue tener un porcentaje grande de vendedores locales, particulares y comercios pequeños en vez de tener un mercado dominado por grandes corporaciones, las prácticas de cosecha y transporte tienen que ser necesariamente óptimas por la alta competencia que supone lo anterior.