Universidad Nacional de Quilmes

Departamento de Ciencia y Tecnología
Tecnicatura Universitaria en Programación Informática

Chasqui: herramienta digital de comercialización para la Economía Social y Solidaria

Leonardo Palazzo

Trabajo de Inserción Profesional
Directora: María Nieves Dalponte A.

Resumen

Chasqui es una herramienta que facilita la comercialización electrónica a organizaciones asociativas de comercialización y consumo de productos de la Economía Social y Solidaria (ESS). Esta herramienta consta de diferentes piezas de software

- lógica de negocio (*backend*)
- aplicación móvil
- panel de administración (*backoffice*)
- Aplicación Web (*frontend web*)

En el presente trabajo se describe el desarrollo de la aplicación *frontend web* para la plataforma **Chasqui**.

Agradecimientos

Ante todo agradezco a la Universidad Nacional de Quilmes por el espacio brindado para poder crecer a nivel humano y profesional, en especial a Mara Dalponte (directora de este Trabajo de Inserción Profesional) por darme la posibilidad de ser parte de este gran proyecto.
Asimismo a todos los docentes que he conocido a lo largo de este camino como parte de la comunidad universitaria. También quiero agradecer al grupo de trabajo conformado por el equipo de Chasqui, quienes continuar con el desarrollo y mantenimiento del proyecto.

Finalmente, agradezco a mi familia por el apoyo brindado durante esta experiencia.

Índice general

Capítulo 1

Contexto

1.1. ¿Qué es la Economía Social y Solidaria?

La ESS tiene sus raíces en la expansión de iniciativas socioeconómicas autónomas de los sectores populares y sus organizaciones de apoyo, como respuesta social a los crecientes niveles de pobreza, exclusión social y precariedad laboral del mundo actual.

Esas iniciativas sociales han impulsado emprendimientos socioeconómicos como opciones de trabajo, ingresos y búsqueda de mejorar la calidad de vida de sus comunidades de pertenencia, que poseen una matriz identitaria de atributos compartidos, entre los que se destaca el desarrollar actividades económicas con una definida nalidad social (en términos generales, mejoramiento de las condiciones, ambiente y calidad de vida de sus propios miembros, de algún sector de la sociedad o de la comunidad en un sentido más amplio), a la vez que implican elementos de carácter asociativo y gestión democrática en un contexto de autonomía tanto del sector privado lucrativo como del Estado.

Algunos ejemplos de estas experiencias son los microemprendimientos, las empresas recuperadas por sus trabajadores, el cooperativismo de trabajo, las formas de intercambio equitativo (mercado solidario, monedas sociales, etc), los microcréditos y las iniciativas de inserción social como las empresas sociales.

1.2. ¿Porqué Chasqui?

Chasqui tiene la funcionalidad básica de cualquier aplicación de compra en línea del mercado tecnológico, pero además aporta características propias que justifican la búsqueda de una identidad a través del desarrollo a medida.

La mayoría de las aplicaciones de compra poseen un foco agresivo en la venta de productos, es decir que muchas veces el cliente compra algo que no es aquello que necesita sino lo que el comercio quiere vender. Además invisibiliza el origen de los productos (quién los produce y en qué condiciones). Como contraparte Chasqui hace foco en el consumo responsable y el trabajo justo brindando herramientas tanto al cliente como a la comercializadora para lograrlo.
Se destacan las siguientes características:

- Visibilización del productor y el proceso productivo a través de dispositivos de comunicación denominados **sellos**. Los productos y productores son mostrados con diferentes niveles de detalle e identificados con características (sellos) como Orgánico, Agroecológico, Cooperativa, Reciclado.

Figura 1.1: Sellos

- Mecanismos de consumo organizado (por ejemplo compra colectiva): este tipo de compra permite a los clientes agruparse para construir un pedido en conjunto y así hacer compras de mayor volumen, algunas veces para alcanzar un monto mínimo de compra o mejorar el precio. Además, permite a la comercializadora una mayor eficiencia a la hora de diseñar la entrega de productos.

1.3. Metodología de trabajo

Contexto

Como se ve en la figura 1.2 de la página 9, el presente trabajo es un módulo de software, que junto a otros compone una aplicación mayor que recibe el nombre *Chasqui*. A continuación, un detalle del alcance funcional de los módulos:

- Lógica de negocio (*backend*): este módulo está oculto al usuario final, sus responsabilidad es resolver la lógica de negocio y exponerla

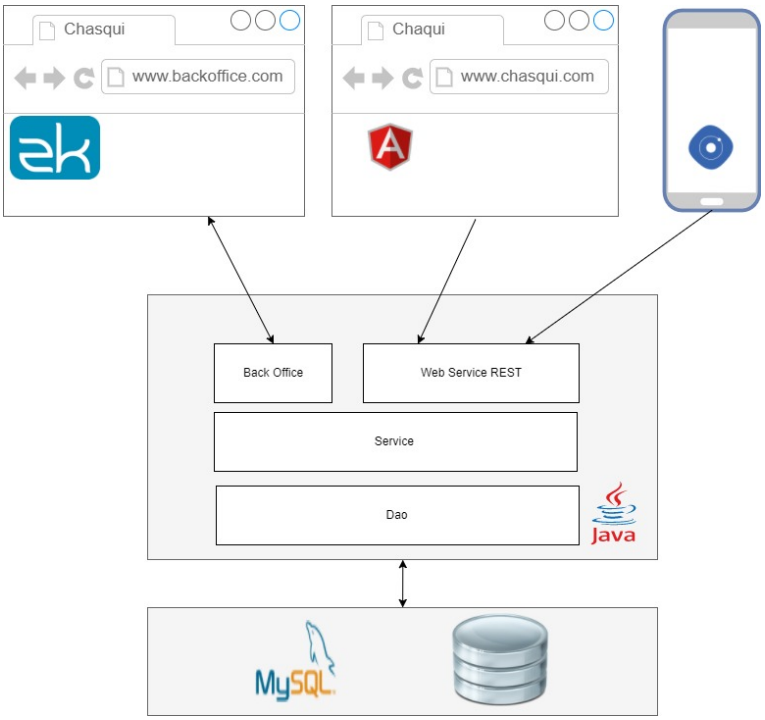

Figura 1.2: Componentes de la plataforma *Chasqui*

mediante servicios *REST* [1]. Se encarga de interactuar con bases de datos, verificar manejos de sesiones de usuarios, enviar mails, comunicarse con otros servicios, entre otros.

- Panel de administración (*backoffice*): esta aplicación es usada por los vendedores (comercializadoras de la economía social, o productores) para llevar a cabo tareas de gestión, principalmente para cargar productos, mantener la lista de precios, ver y administrar los pedidos.

- Aplicación Web (*frontend web*): es una página web capaz de correr en un navegador con la cual el cliente podrá navegar el catálogo de productos, realizar los pedidos de productos, gestionar sus grupos de compra colectiva, los datos de su perfil y ver sus notificaciones.

- Aplicación móvil (*Mobile App*): es una aplicación para dispositivos móviles con funcionalidad análoga a la anterior.

[1] REST no es un estándar, simplemente define unos principios de arquitectura a seguir para implementar aplicaciones o servicios en red. Sin embargo, REST se basa en estándares para su implementación: HTTP, XML, etc. Los servicios REST tienen las siguientes características: Cliente-Servidor, Sin estado, Información cacheable, Interfaz uniforme, Acceso a recursos por nombre, Recursos relacionados, Respuesta en un formato conocido.

A pesar de que los módulos, de manera individual, no cumplen los objetivos de *Chasqui* por estar funcionalmente acoplados, una buena arquitectura es la que hace a los módulos técnicamente independientes (ver figura 1.2), y en este caso cada módulo está implementado en diferentes tecnologías comunicadas a través de servicios REST.

A partir de la arquitectura descrita y que los restantes módulos fueron desarrollados por otros equipos, uno de los desafíos más importantes fue buscar un mecanismo de trabajo distribuido eficiente tanto con los desarrolladores como con los encargados del diseño y definición de funcionalidades. La metodología debía contemplar factores importantes entre los cuales está el hecho de no compartir el espacio de trabajo diario ni poder unificar calendarios.

Comunicación

Se tomó como base la metodología Scrum [**?**], por lo que se tenía una reunión presencial en la cual se consensuaba funcionalidades (*Product Backlog*) y se priorizaban las próximas tareas a realizar (*Sprint Backlog*). Este ciclo se repetía cada 6 semanas en promedio (ver figura 1.3).

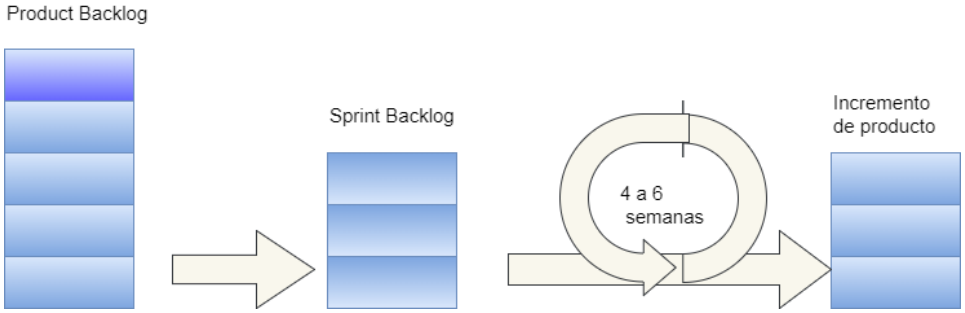

Figura 1.3: Abstracción de la metodología Scrum utilizada

Habitualmente este tipo de metodología contempla llevar a cabo reuniones diarias que no podíamos cumplir, sin embargo se reforzó la comunicación asincrónica por medio de otras herramientas. Gran parte de la comunicación fue a través de la aplicación *Trello* (ver figura 1.4), que además de comunicar permite una organización de las tareas de una forma muy parecida a un tablero físico. También en varias oportunidades se tuvo videoconferencias para consensuar algún tema técnico o funcional.

En este tipo de arquitectura y equipos distribuidos un factor fundamental es la comunicación de documentación de servicios disponibles, donde se brinda información sobre qué métodos están disponibles para serr utilizados, qué parámetros reciben, qué datos devuelven, etc. Con este fin se utilizó la herramienta para documentación de API llamada *Apiary* [**?**](ver figura 1.5).

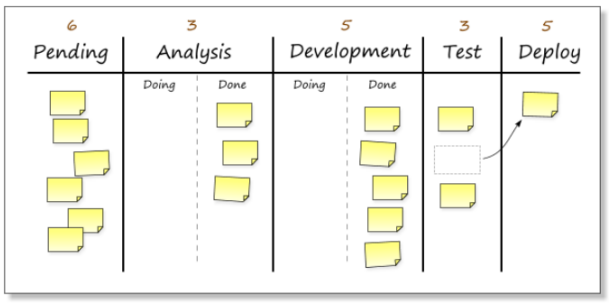

Figura 1.4: Imagen ilustrativa de tareas organizadas en un tablero

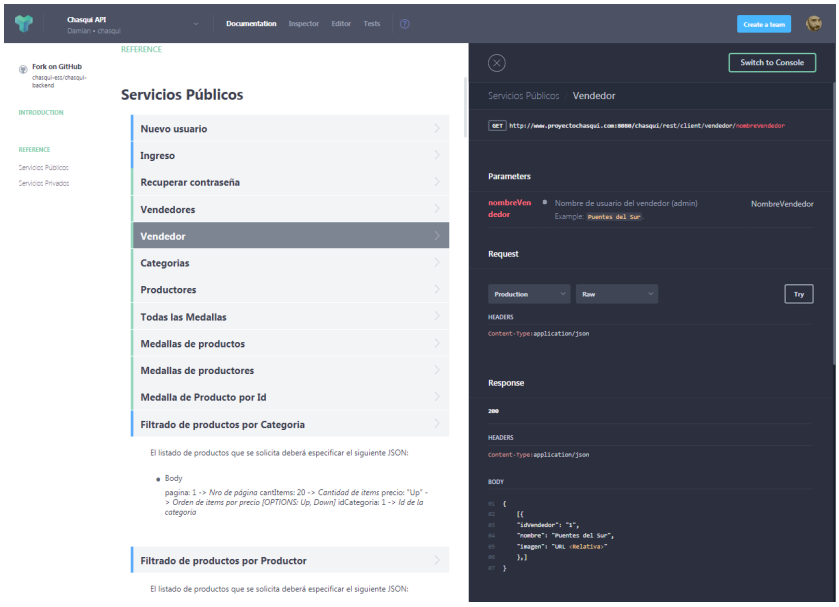

Figura 1.5: Documentación de Servicios REST mediante Apiary

Backend Simulado

Al comienzo del presente proyecto, el desarrollo del *backend* y del *backoffice* estaba ya avanzado y estaban definidas varias funcionalidades. Muchas estaban implementadas en el módulo de *Backoffice*, pero aún no se habían expuesto los servicios que debían ser consumidos por el *frontend*. Estos factores, sumado a que no estaban definidos los tiempos de cada desarrollo, fue necesario plantear una metodología técnica que no estableciera una dependencia entre ambos trabajos. De esta manera surgió el desafío de abordar un esquema de trabajo donde los diferentes módulos puedan ser desarrollados de forma independiente y hacia el final del trabajo se integren con el menor costo de implementación posible.

Con el fin de independizar el desarrollo se creó un módulo temporal (*mock*) para simular los servicios del *backend* real, mientras éste estaba en desarrollo. La definición de estos servicios temporales partió del modelo de entidades que el backoffice ya tenía en ese momento y de los bocetos de pantallas. A medida que el desarrollo de los servicios reales avanzaba se ajustaban los servicios mocks o bien directamente se reemplazaban las llamadas a los servicios reales. Muchas veces los servicios mock fueron utilizados como referencia para la implementación de los servicios reales.

Bocetos de las pantallas

Para los diseños de pantalla en la primera etapa se recibió una descripción funcional y bocetos de las pantallas que permitieron comenzar a trabajar en el maquetado de las mismas (ver figura 1.6)

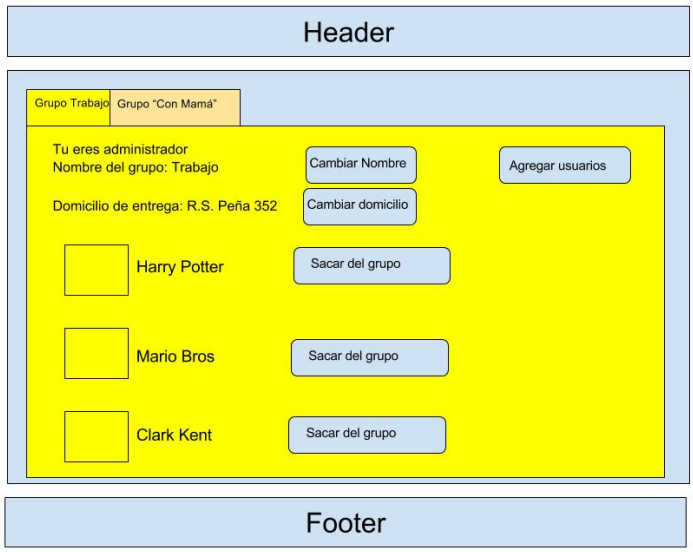

Figura 1.6: Boceto pantalla de compra colectiva

Entrega y validación

La entrega consistió en plasmar los cambios en el repositorio y asegurar que que tenga la documentación necesaria para poder correr la aplicación. Estos cambios eran desplegados en un servidor de pruebas para que los referentes de productos puedan hacer las pruebas de los casos de uso. A tal fin se documentó el proceso de configuración y despliegue como se describe en el apéndice **??**. El medio para comunicar el reporte de errores fue el tablero de *Trello*.

Capítulo 2

Funcionalidad

2.1. Registro e ingreso

La aplicación puede ser navegada de forma anónima, en cuyo caso solo podrá acceder a secciones que no impliquen una transacción, como por ejemplo, ver la página inicial, las características de productores o ver el catálogo de productos (ver figura 2.1).

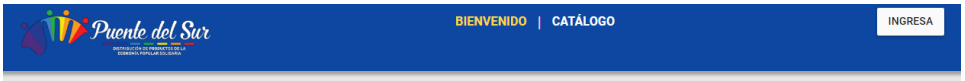

Figura 2.1: Opciones de menú de la navegación anónima

Para poder hacer un pedido, el usuario deberá registrarse ingresando usuario, contraseña, y una serie de datos personales entre los cuales se destaca el domicilio donde pretende que se le entreguen los productos. Al finalizar los formularios de registro se le enviará un correo para validarlo. Luego podrá acceder a todas las funcionalidades por medio de usuario y contraseña (ver figura 2.2). También existe la posibilidad de recuperar la contraseña en caso de que el usuario la olvide.
Una vez iniciada su sesión, el usuario tiene disponible un menú con mayor funcionalidad (ver figura 2.3).

2.2. Características de productos y productores

En esta sección, que es puramente informativa, podremos conocer a los productores cuyos productos se venden en el catálogo. El vendedor es quien define la información a mostrar y la intención es que visibilice la historia, quienes son, cuál es su territorio, cómo se organizaron y sus valores (ver figura 2.4) .

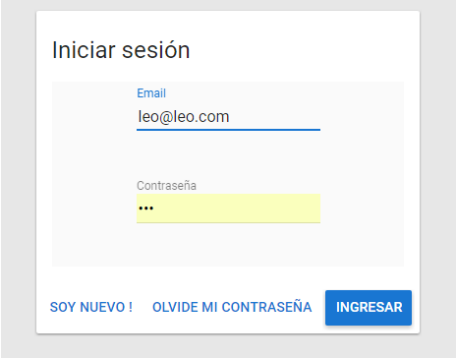

Figura 2.2: Formulario de inicio de sesión

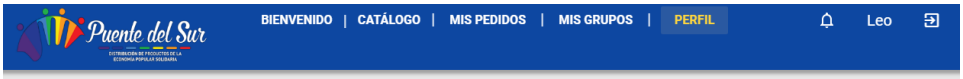

Figura 2.3: Menú de usuario con sesión iniciada

2.3. Catálogo y búsquedas

En esta sección (ver figura 2.5) el consumidor podrá encontrar los productos ofrecidos de manera paginada y previsualizados con los datos principales: nombre del producto, precio de venta, nombre del productor y sellos identitarios. Además, es posible filtrar los productos por distintos criterios (ver figura 2.6), a saber: por productor, por sello , por la categoría a la que pertenecen o bien un por texto que coincida parcialmente con el nombre o sello del producto.

Por otro lado, desde esta pantalla se podrá agregar el producto a un pedido en curso (esto es, en estado abierto), y es importante destacar que se puede tener más de un pedido abierto al mismo tiempo, dado que es posible mantener en paralelo distintos **contextos de compra**, por ejemplo asociados al pedido individual y a pedidos en los **grupos de compra colectiva**. Para esto se diseñó un mecanismo que permite cambiar el contexto de compra (ver figura 2.7) para agregar productos, y que también cambia adecuadamente el pedido en la previsualización (ver figura 2.5 a la izquierda debajo de los filtros).

2.4. Perfil del usuario

En esta pantalla (ver figura 2.8) el usuario podrá modificar sus datos personales, asociar más de un domicilio , ver sus notificaciones, e inclusive cambiar su contraseña (ver figura 2.9).

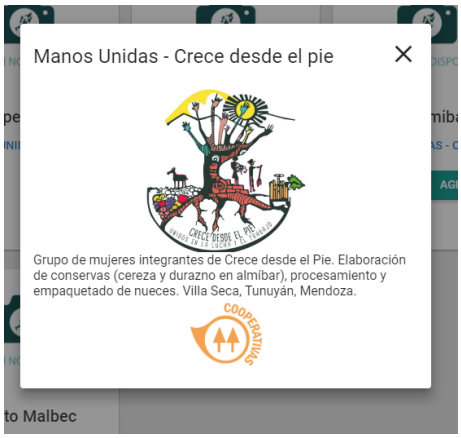

Figura 2.4: Información sobre el productor.

2.5. Modos de compra

Compra Individual

El modo de compra individual en Chasqui es el que habitualmente encontramos en la mayoría de las tiendas en línea. El usuario se registra, agrega los productos que desea en un carro de compras, y finalmente confirma el pedido eligiendo un domicilio de entrega. A partir de entonces el vendedor tiene disponible el pedido en el **backoffice**.

Compra colectiva

A diferencia de otros sistemas de compra en linea, Chasqui soporta la compra colectiva, donde los usuarios pueden crear grupos para comprar colectivamente (ver figura 2.10).

Cada consumidor realiza la compra como si se tratara de una compra individual, habiendo elegido el contexto de compra correspondiente al grupo deseado (ver sección 2.3), y de esta manera los productos que elige quedan asociados a un mismo pedido colectivo. Todos los miembros del grupo tienen la posibilidad de ver el pedido individual realizado por los otros miembros del grupo de compras colectivas.

El usuario que da de alta el grupo tendrá el rol de administrador. Esto implica que será el único habilitado para invitar otros usuarios a formar parte del grupo, a través de la cuenta del mail de cada invitado. Además es el único que puede abrir un pedido y confirmar el pedido colectivo.

El pedido colectivo llegará al domicilio que configuró el administrador, con

las compras de cada usuario debidamente identificadas. El responsable abona en efectivo el total de la compra colectiva en el momento de la entrega. Esta lógica de consumo organizado provee un mecanismo más efectivo para la logística de distribución de la comercializadora.

En la figura 2.10 se observa:

1. el usuario tiene dos grupos de compra, 'familia' y 'trabajo'.
2. en este caso tiene foco en 'familia' del cual es administrador.
3. sobre los integrantes del grupo:
   a) A la izquierda se ve que el usuario `a@a.com` no ha aceptado la invitación al grupo, por lo tanto a la derecha no aparece.
   b) Por el contrario el usuario `armando@a.com`, sí ha aceptado la invitación al grupo, por lo tanto a la derecha se puede ver qué productos agregó.

2.6. Notificaciones

El consumidor puede recibir notificaciones en distintas situaciones que se enumeran a continuación (ver figura 2.11), separadas en dos tipos: informativas y confirmables.

- Notificaciones informativas: son aquellas notificaciones que pueden estar en uno de dos posibles estados: leída y no leída. Ejemplo de este tipo de notificaciones son:
  1. Avisos de fecha de entrega
  2. Vencimiento de un pedido
  3. Nuevo pedido en un grupo de compras colectivas
  4. Confirmación del pedido colectivo
- Notificaciones confirmables: La confirmación de estas notificaciones disparan alguna acción en el servidor. Esto significa que pueden estar en uno de dos estados: aceptada o rechazada. Ejemplos de esto son:
  1. Invitación a formar parte de un grupo de compras colectivas
  2. Invitación a coordinar un grupo de compras colectivas

2.6.1. Multicatálogo

Cada organización deberá poder administrar y mostrar su catálogo de forma totalmente independiente. Esto implica que debe tener la flexibilidad de poder configurar por organización determinadas características de negocio y estética. [1]

[1] incompleto?? quizás sacaría esta sección. Está fuera del alcance del TIP. Salvo que hayas tomado decisiones teniendo en cuenta el contexto multicatalogo

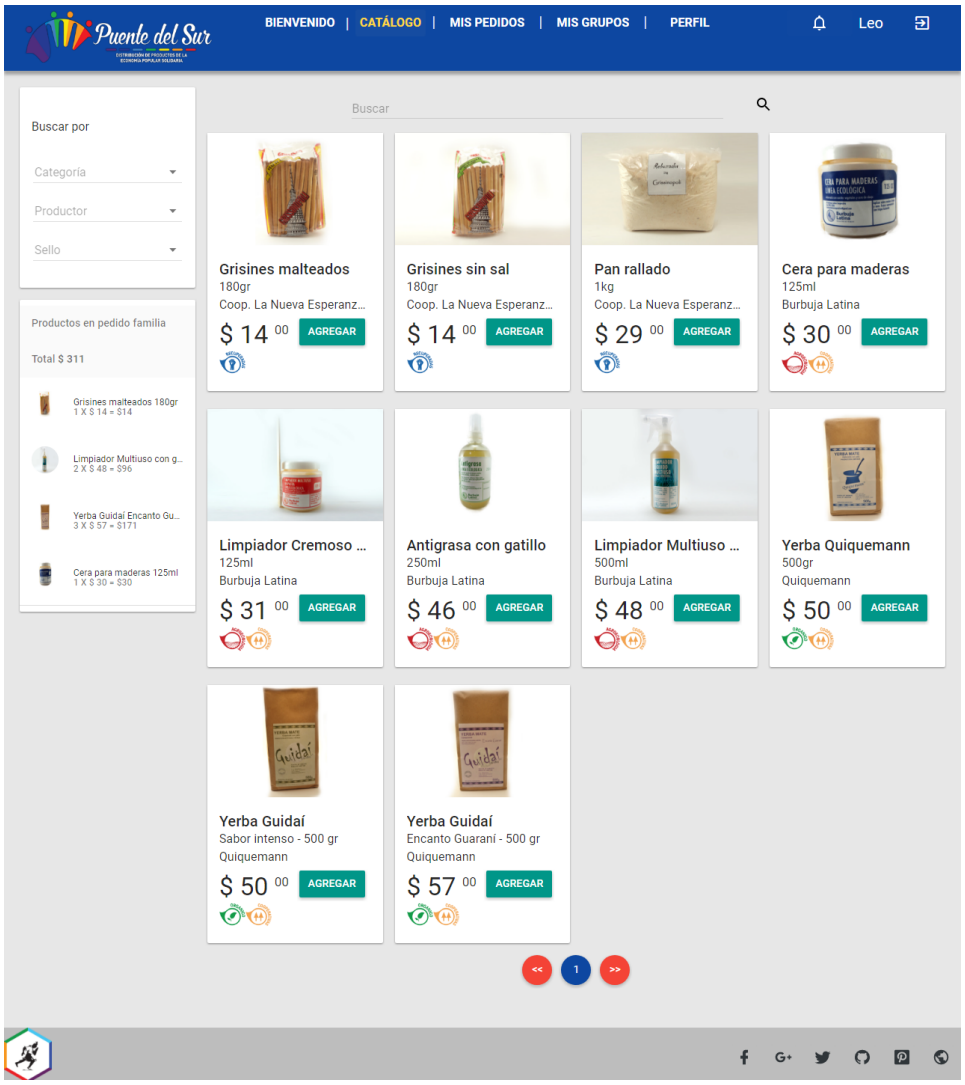

Figura 2.5: Página de búsqueda de productos con el contexto de compra

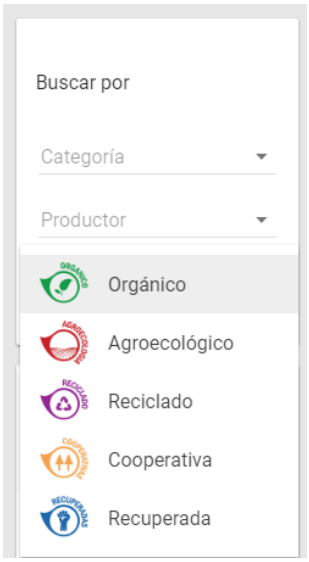

Figura 2.6: Filtro por sellos

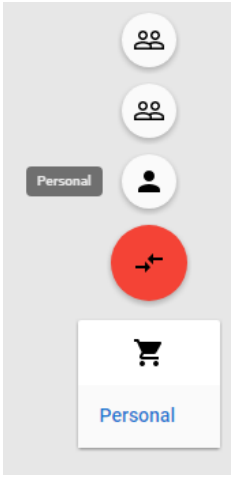

Figura 2.7: Visualización de contextos de Compra

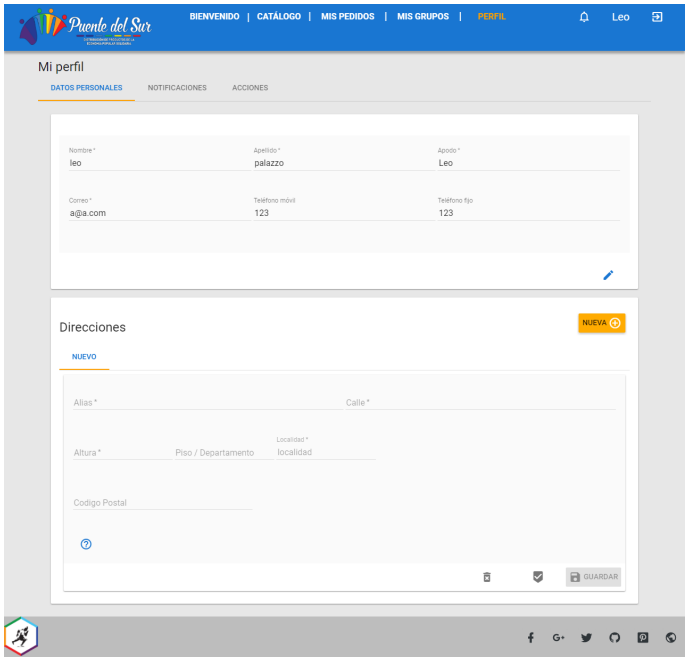

Figura 2.8: Pantalla de edición de perfil de usuario

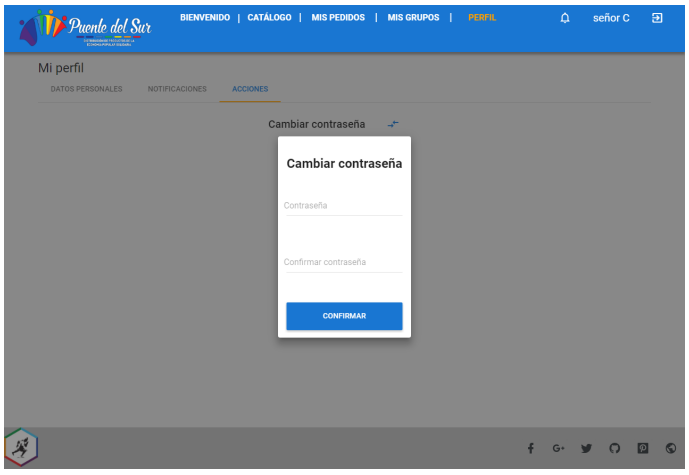

Figura 2.9: Pantalla de cambio de contraseña

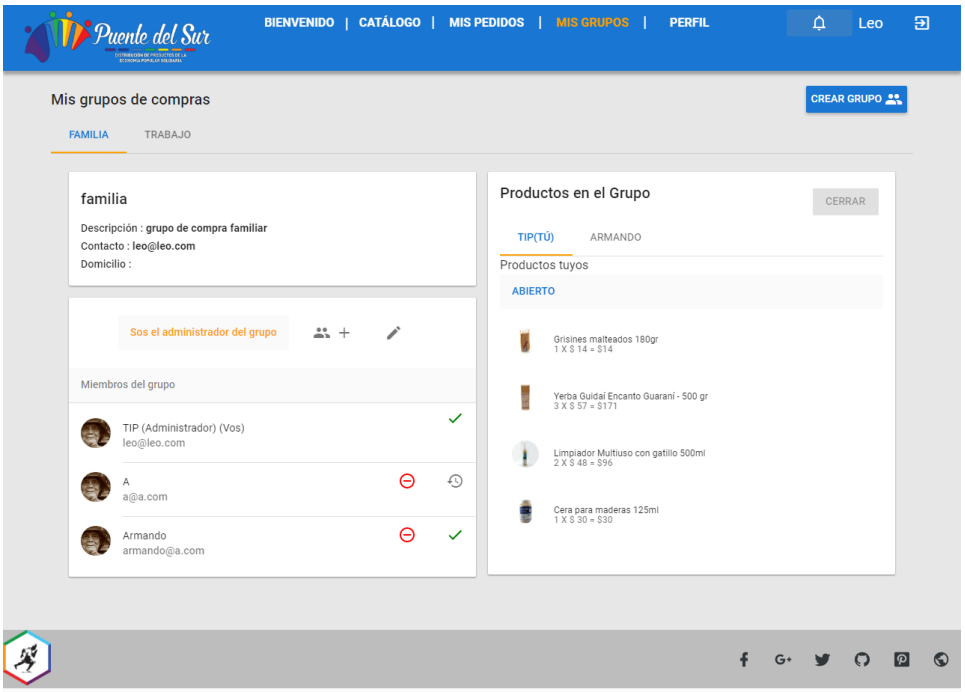

Figura 2.10: Pantalla de administración de grupos de compra colectiva

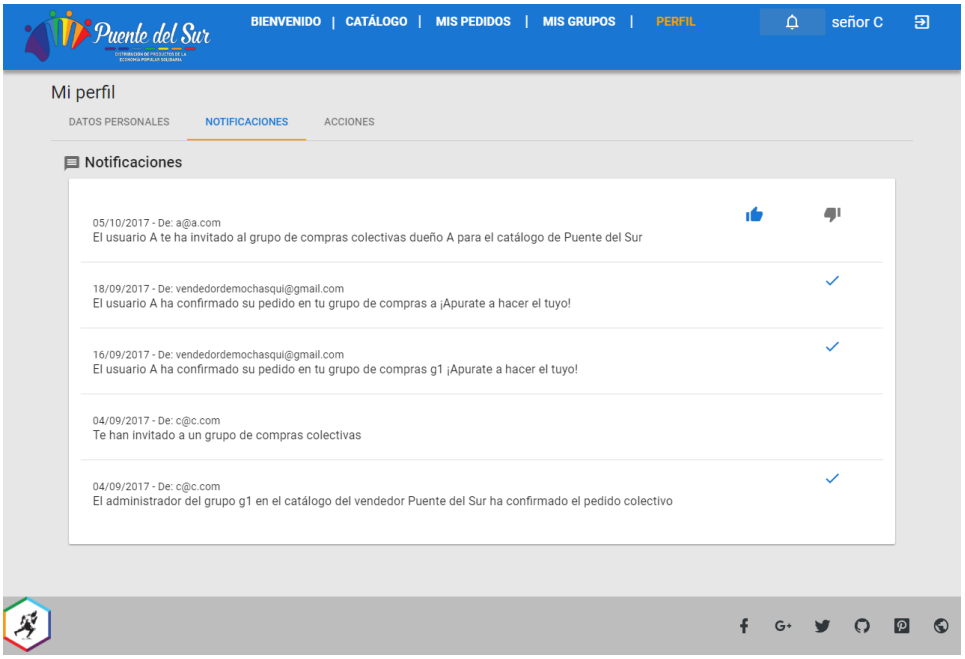

Figura 2.11: Pantalla de notificaciones del usuario

Capítulo 3

Implementacion

3.1. Tecnologías utilizadas

Lenguaje de programación

La plataforma Chasqui posee una aplicación servidor (backend) que expone su lógica de negocio a través de servicios web de tipo REST, y por lo tanto la interfaz gráfica que se desarrolló en el presente trabajo es en si misma una aplicación web que consume los servicios mencionados.

Tomando esto como precondición a la hora de definir la arquitectura para este desarrollo (de ahora en más, *frontend*), se descartaron tecnologías que crean, procesan o sirven la interfaz desde el servidor dinámicamente, a saber: Php, Java Jsp , Jsf , Struts , Spring MVC, ZK , etc. Esto se debe a que las anteriores implicarían una invocación innecesaria a un servidor que a su vez debería invocar a los servicios mencionados. Por lo tanto la primera decisión de arquitectura fue utilizar una tecnología capaz de invocar a los servicios y actualizar la interfaz desde el propio navegador.

A principio de 2016 se analizaron varias opciones para tal fin, y se eligió **AngularJS** [**?**]. AngularJS (comúnmente llamado Angular.js o AngularJS 1), es un framework de JavaScript de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página (SPA [1]) . Angular se eligió por ser estable, tener un equipo que lo mantiene, y ser popular entre los desarrolladores, todos estos factores son de especial interés si se quiere mantener o extender esta aplicación. No se optó por Angular 2 ya que para la fecha todavía estaba en una versión prematura.

[1] Una SPA es una aplicación web que se ejecuta bajo una única página. Su contenido cambia en una determinada porción de la página lo que permite reutilizar, por ejemplo, el encabezado, el pie de página y los menú de navegación.

Para el desarrollo del *Mock*, que es esencialmente un backend, se utilizo **Java**. Debido a que este módulo temporal no es parte del entregable, pues sería descartado una vez que el backend real estuviera terminado, la elección de la tecnología buscó reducir la curva de aprendizaje para tener el menor impacto posible en el tiempo de desarrollo.

Diseño visual

Además de los aspectos puramente funcionales se tuvo que definir la estrategia de trabajo en cuanto al diseño. En un primer momento se evaluó diseñar e implementar componentes de html y css propios, pero esta estrategia suele tener un costo muy alto en tiempo.

Por otro lado se consideró utilizar diferentes componentes de interfaz web [2] de licencia libre que se ajusten a las necesidades concretas y aplicarle un diseño. Esto tiene un costo medio ya que hay que aplicar diseño a los componentes que probablemente no sean todos del mismo autor.

A continuación se analizó la posibilidad de utilizar una plantilla de *E-commerce* paga, con un diseño profesional. Estas plantillas usualmente son una compilación de archivos HTML5 y CSS3 que resuelven cuestiones de diseño tales como combinación de colores, distribución de componentes y fuentes. Sin embargo, en general estas plantillas no están pensadas para ser compatibles con AngularJS.

La alternativa más conveniente era encontrar una biblioteca libre que resuelva diseño e implementación de componentes AngularJs todo en un solo framework. En esta etapa de análisis de posibilidades de diseño web y definición de tecnologías, se encontró que el concepto de *Material Design* [3] estaba marcando tendencia. Orientando la búsqueda a este tipo de diseño fue que se encontró y eligió *Material Angular* [**?**], un framework de licencia libre que implementa componentes de interfaz web basados en los lineamientos de *Material Design*. Esto redujo considerablemente los costos de implementación.

3.2. Componentes de la arquitectura

En la presente sección se describen los componentes de la arquitectura que se definió para el proyecto de *frontend*.

[2] Según *usability.gov*, las interfaces de usuario están compuestas por elementos como selectores de opciones, selectores de fecha, campos para ingreso de texto, carrusel de imágenes, entre otros. [**?**]

[3] Conjunto de normas de diseño desarrolladas por *google* con foco en el sistema operativo *Android*, pero que se expandió a la plataforma web [**?**]

Tomando como referencia las buenas prácticas de estructura de proyecto AngularJs, que están basadas en el patrón de diseño MVC, se separó el código fuente en los grupos que se describen en la figura 3.1.
La estructura carpetas y archivos utilizada es un factor importante para la organización y lectura del código fuente, si bien la buenas prácticas no coinciden necesariamente en una única forma, se opto por la opción de agrupar por componente. El siguiente es un ejemplo de la estructura de un proyecto:

```
app/
----- shared/ --> para componentes reutilizables
---------- sidebar/
--------------- sidebarDirective.js
--------------- sidebarView.html
---------- article/
--------------- articleDirective.js
--------------- articleView.html
----- components/ --> cada componente se trata como una pequeña
-------------------- aplicación angular
---------- home/
--------------- homeController.js
--------------- homeService.js
--------------- homeView.html
---------- blog/
--------------- blogController.js
--------------- blogService.js
--------------- blogView.html
----- app.module.js
----- app.routes.js
assets/
----- img/ --> imagenes e íconos
----- css/ --> archivos de estilo (SCSS o LESS files)
----- js/  --> JavaScript que no es angular
----- libs/--> bibliotecas de terceros como jQuery, Moment,
-------------- Underscore, etc.
index.html
```

3.2.1. Componentes HTML y controladores

Los componentes de alto nivel definidos para el presente desarrollo se construyen mediante un archivo .html que describe la interfaz y un archivo .js que contiene la lógica para controlar la interfaz. A continuación se enumeran diichos componentes:

- Sección para administración de perfil de usuario
- Sección para administración de notificaciones
- Formulario de usuario

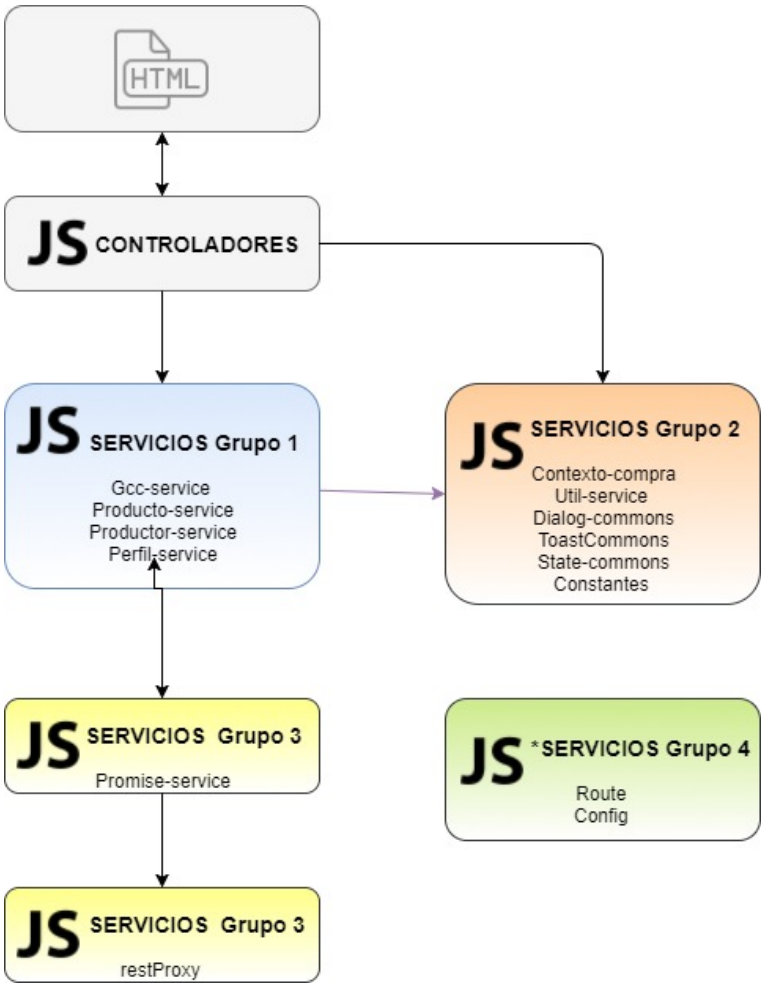

Figura 3.1: Arquitectura: grupos de servicios

- Formulario de dirección
- Formulario de grupo de compra
- Formulario de login
- Sección para administración de grupos
- Ficha de datos de grupo
- Sección para administración de pedidos
- Ficha de datos de pedido
- Sección para ver el catalogo de pedidos

- Ficha de dato de producto, con botón de compra
- Opciones de filtrado de catalogo
- Ficha de datos de productor
- Ficha de datos de sellos
- Lista de productos minimizada para previsualización
- Información para página de inicio
- Menú
- Cabecera de página
- Pie de página

3.2.2. Hojas de estilo CSS

Como se utilizó la herramienta *Material Angular* que provee estilos, se redujo notablemente el uso de hojas de estilo. Sólo se utilizó en casos particulares para ajustar aspectos que el framework no contempla o bien sobre-escribir algún diseño.

3.2.3. Servicios

En AngularJS los servicios son objetos de tipo *singleton*, inyectables por *Dependency Injection* [4], donde se define la lógica de negocio de la aplicación, con el objetivo de que sea reutilizable e independiente de las vistas.

Los servicios definidos se clasifican en cuatro grandes grupos: servicios de lógica de negocio, servicios transversales (comúnmente requeridos en toda la aplicación), servicios de apoyo para la comunicación con el backend y servicios que son parte de las estructura necesaria de Angular.

Servicios de lógica de negocio

La responsabilidad de este grupo de servicios es fundamentalmente ser el medio de comunicación con el backend, proveyendo un mecanismo simple de acceso y envío de información. Si bien es normal describir esta capa como de lógica de negocio, al tratarse de un frontend el código de este tipo tiende a ser

[4] En programación, inyección de dependencias (en inglés *Dependency Injection*, DI) es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree el objeto. El término fue acuñado por primera vez por Martin Fowler.[**?**]

la tendencia a tener estque una correspondencia con los servicios del *backend*, puese tipo de minimalista, apencódigo es cero ya que se espera que sea el backend quien tenga es responsabilidad.

- **ciona con la lógica de GCC-service**: este subgrupo de servicios refieren a la parte del domino relacionada con los grupos de compra, por ejemplo: crear un grupo de compra, invitar un usuario al grupo, crear un pedido y aceptar una invitalción. i

- **producto-service**: este subgrupo de serviciaos refiere a la parte de domino relacionada con los productos, por ejemplo: ver categorís, ver medalla, buscar productos, cancelar pledido.

- fperfil-service: este subgrupo de servicios refiere a la parte de dominio relacionada el usuario, por ejemplo: ver sus direcciones, ver sus notificaciones, marcar una notificación como leída, cambiar contraseña, iniciar sesión y crear un usuario.

Servicios transversales

- **contexto-compra**: Tiene como responsabilidad tener el contexto de compra del usuario para mantener la coherencia en la navegación. Por ejemplo : qué producto seleccionó, qué pedido seleccionó, qué grupo de compra selecciono.
  Con el objetivo de evitar llama innecesarias a los servicios mientras el usuario navega y así mejorar el rendimiento, se implemento una ***caché*** para los servicios que traen la información de grupo y productos del usuario. Esta caché tiene un tiempo de expiración definido en el servicio de constantes que se puede ajustar según se crea conveniente. Cabe aclarar que este tipo de problema generalmente no se detecta en etapa de desarrollo sino hasta que se empiezan con las pruebas de rendimiento o en el peor de los casos cuando la aplicación está productiva. En el presente proyecto se eligió prevenir un potencial problema a futuro.

- **util-service**: aquí se disponen servicios útiles como verificar si una cadena está vacía sin preocuparse si en *null* o *undefined*, verificar si una variable es *undefined* o es *null*, formateo de fecha, traducción mediante i18n (ver sección 3.2.4).

- **dialog-commons**: este servicio permite crear de forma rápida y simple ventanas con un mensaje que debe ser confirmado por el usuario, también *pop-up* con mensaje y opción de cancelar o confirmar una acción.

- **toast-commons**: este servicio permite enviar un mensaje al usuario de forma rápida y sencilla. A diferencia de un dialogo, este es visualmente más chico, aparece desde los margenes de la pantalla y usualmente no requiere confirmación.

- **state-commons**: este servicio es un nivel de abstracción para el manejo de persistencia en el navegador. Entre otras cosas se utiliza para mantener estados del usuario como por ejemplo : el *token* de sesión, datos básicos del usuario, última pantalla navegada, producto seleccionado, y cualquier otro dato que sea relevante mantener comentarioindependientementea que te refiris con esto? de la navegación, inclusive si cierra la aplicación y vuelve a entrar más tarde.

- **Constantes**: servicio que provee los valores de las constantes tales como URLs de los servicios REST, tiempo de duración de cache de datos y el tiempo de intervalo de refresco de notificaciones, entre otros.

Servicios para comunicación por REST al backend

Se diseñaron las siguientes capas para optimizar y modularizar el código de la invocación a los servicios REST.

- **promise-service**: es una capa intermedia que se utiliza antes de llamar a los servicios REST, cuyo objetivo es simplificar el uso de *promises*[5], que es una forma útil de manejar procesos asincrónicos. El principal objetivo es reducir la cantidad de código y mejorar su lectura.

- **restProxy**: es una capa de abstracción del componente de Angularjs `$http` para llamar a servicios REST, que define métodos para hacer invocaciones *Get* y *Post* de los servicios tanto públicos como privados. En este último caso se necesita crear encabezados usando el *token* del usuario según convención establecida con el equipo de desarrollo del *backend*. Se destaca además el registro por log de cada acción y las funciones de manejo de errores comunes, donde puede extenderse el comportamiento para manejar nuevos errores.

De esta manera, con ayuda de estas dos capas de servicios evitamos repetir el mismo código en cada llamada al un servicio REST del *backend*, en la figura 3.2 se muestra un ejemplo de solo dos invocaciones a diferentes servicios donde se ve que la única diferencia es la URL del servicio. Para entender la magnitud se debe tener en cuenta que se realizan cerca de *50 invocaciones a servicios* [6] .

[5] Las promesas son objetos que facilitan la gestión de la programación asíncrona dentro de JavaScript. [**?**]

[6] `en cuanto tiempo?`

```
var productoDestacadoService = function getInformacion () {
        var defered = $q.defer();
        var promise = defered.promise;

        $http.get('http://chasqui.com/productos-destacados')
            .success(function(data) {
                defered.resolve(data);
            })
            .error(function(err) {
                defered.reject(err)
            });

        return promise;
    }

var productoCategoriaService =  function getInformacion (categoria) {
        var defered = $q.defer();
        var promise = defered.promise;

        $http.get('http://chasqui.com/productos-by-categoria/'+categoria)
            .success(function(data) {
                defered.resolve(data);
            })
            .error(function(err) {
                defered.reject(err)
            });

        return promise;
    }

productoCategoriaService.getProductosByCategoria('vinos')then(callback).catch(errorCallback);
productoDestacadoService.getInformacion().then(callback).catch(errorCallback);
```

Figura 3.2: Ejemplo de dos invocaciones a servicios

Servicios estructurales de Angular

- **route**: Define la relación entre los archivos html, sus controladores, la url a la que responden. Además define, entre otras cosas, qué tipo de parámetro puede recibir cada página.

- **config**: Se definen configuraciones de componentes de terceros como logs, dialog, idioma , progress bar.

3.2.4. Internacionalización y localización

Internacionalización (I18N) es el diseño y la preparación de software, sitios web y otras aplicaciones a fin de que el código de origen sea válido para la cultura local y pueda soportar varios idiomas y escrituras.[?]

Para poder lograr esto se definen los textos en archivos con el formato clave y valor en un archivo del tipo *json* en la carpeta *locale* y en la subcarpeta correspondiente al idioma , por ejemplo `en` para ingles y *es* para español. Además para una mejor lectura se agrupan en archivos según pantalla o módulo.

Para poder usarlo desde código html se debe usar el atributo *translate*, y desde un código js se usa el servicio *translate* implementado especialmente

para tal fin (ver figura 3.3)

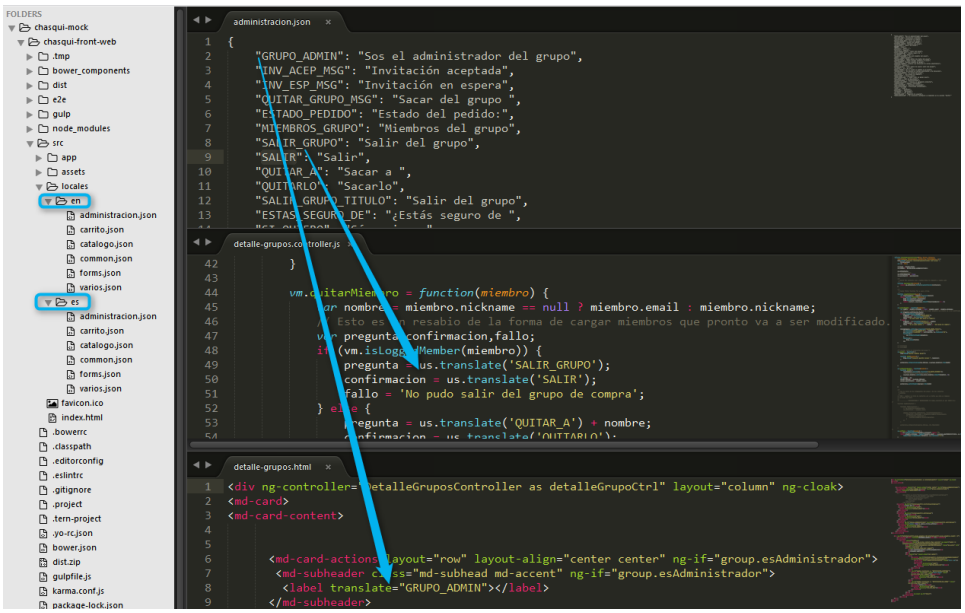

Figura 3.3: Ejemplo de i18N

3.3. Componentes de la arquitectura Mock

En esta sección de describe brevemente, por tratarse de un proyecto temporal, la arquitectura usada en el *Mock* (ver figura 3.4) .

Estructura del proyecto

Se divide en tres grupos

- Model: Como punto de partida se tomo el modelo del Backend en ese momento
- Servicio : Emula la lógica de negocio y devuelve los datos de negocio.
- Controller En esta capa se exponen los servicios.

Dependencias utilizadas

- springframework para injección de dependencias y exposición de los servicios REST mediante anotaciones.

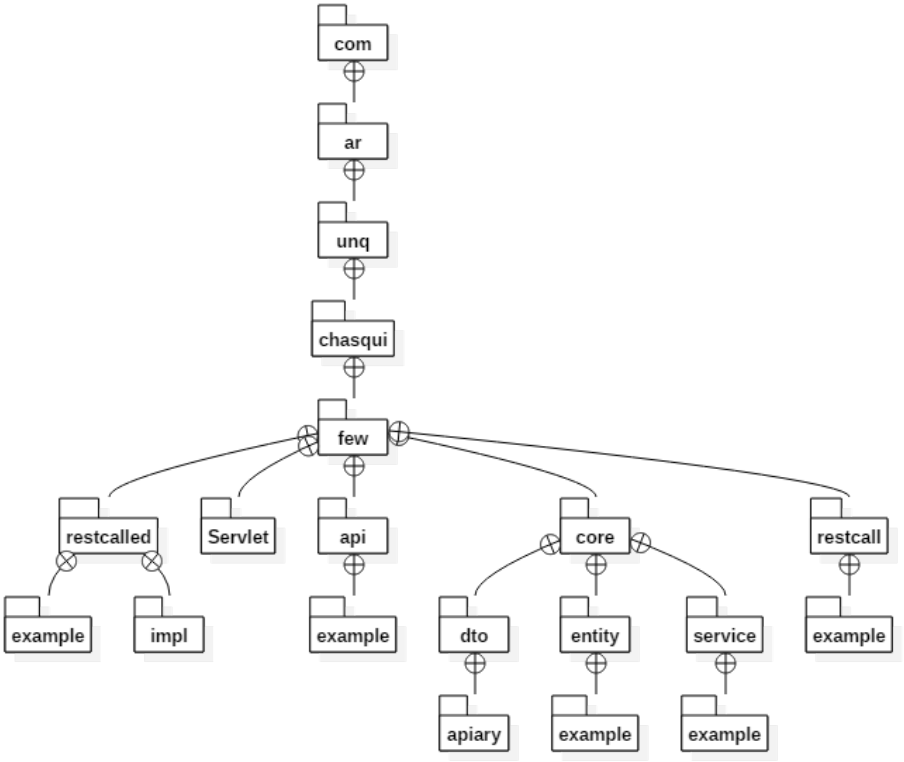

Figura 3.4: Diagrama de paquetes.

- jackson para la transformación de texto en formato *json* a entidades *java* y viceversa.

Capítulo 4

Evaluación del desarrollo

4.1. Retrospectiva

Elegir un tema para el trabajo final de la carrera se podría decir que es un trabajo en si mismo, no es fácil, hay que entender qué es lo que a uno lo motiva para tener éxito. Las motivaciones pueden ser la investigación, el desarrollo con nuevas tecnologías o, como en mi caso, simplemente crear. Luego de algunos intentos fallidos de elección de tema para el trabajo, apareció la oportunidad de formar parte del equipo de Chasqui y no lo dude. Unirme me dio la oportunidad de conocer un enfoque diferente de un modelo de negocio como ser el caso de la ESS, y más aún poder contribuir al movimiento. La mayor satisfacción personal de este trabajo será ver la herramienta funcionando y brindando soluciones a las cooperativas.

Por otro lado, el principio de un proyecto no significa escribir código el primer día, pues se comienza teniendo reuniones periódicas para conocer al resto del equipo, entender el modelo de negocio, comprender la percepción del producto por parte de sus referentes, etc. Es decir: descubrir qué hay que hacer. Además, formar parte de un equipo distribuido que no esta cien por ciento avocado al proyecto genera problemas de calendario que terminan dilatando los tiempos ideales.

La segunda etapa consistió en definir el diseño, la arquitectura, las herramientas, y en el contexto de que el proyecto seguirá creciendo una vez finalizado el trabajo, tener la libertad al tomar estas decisiones es gratificante. En la industria no siempre se puede, pero a la vez tiene una cuota de condicionamiento saber que las decisiones que se toman en esta etapa es realmente costoso cambiarlas una vez que el proyecto maduró. Por lo comentado, esta etapa tomo un tiempo importante de investigación y evaluación.

Finalmente comenzó la etapa de desarrollo, que no tuvo mayores incon-

venientes técnicos. Si bien este no fué un trabajo de excesiva complejidad técnica, es un trabajo largo que requiere empeño, y que va más allá del desarrollo: requiere ser polivalente. Los tiempo de desarrollo fueron intermitentes, hubo etapas donde se avanzo mucho y etapas que por razones personales, por algún tipo de dependencia del resto del equipo, por cambios funcionales o de alcance, se avanzo poco. Como resultado el proyecto se extendió mas de lo planeado, y para hacerlo evidente: el proyecto tiene más de 400 *commits* en el repositorio.

Destaco que toda la experiencia adquirida en el proceso del trabajo, las buenas y malas, son cotidianas en la industria, lo cual valoro en lo personal e inclusive recomiendo para alumnos que todavía no hayan dado sus primeros pasos en esta.

4.2. Trabajo a futuro

Durante toda la etapa del desarrollo se detectaron distintas mejoras que por competir con otras prioridades quedaron fuera del alcance del TIP.

Algunas de ellas son:

- Estrategias de prueba. Tener test unitarios, test de integración, integración continua, para mejorar la calidad del producto.
- Hacer una iteración de pruebas y arreglos con un equipo fuerte de QA para mejorar la calidad.
- Mejorar la documentación técnica para futuros integrantes del equipo.
- Hacer que el *frontend* se adapte según las estrategias de comercialización definidas para cada organización

A la fecha de finalización del presente informe se relevaron 34 nuevas funcionalidades y 84 oportunidades de mejora.

Bibliografía

[1] Williams Stallings, *Computer Organization and Architecture*, octava edición, Editorial Prentice Hall, 2010.