

INSTITUTO TECNOLÓGICO DE TIJUANA

Ingeniería en Sistemas Computacionales



DISEÑO Y DESARROLLO DE SISTEMA MULTI-PLATAFORMA PARA
GESTIÓN DE VENTAS Y LOGÍSTICA DE MICROEMPRESA.

POR

JOSÉ LUIS MURILLO RÍOS

PARA OBTENER EL TÍTULO DE INGENIERO EN SISTEMAS
COMPUTACIONALES

TIJUANA, B.C.

AGOSTO 2019

DISEÑO Y DESARROLLO DE SISTEMA MULTI-PLATAFORMA PARA
GESTIÓN DE VENTAS Y LOGÍSTICA DE MICROEMPRESA.

POR

JOSÉ LUIS MURILLO RÍOS

AGOSTO 2019

RESUMEN

Cada día mas emprendedores inician su negocio con la ayuda de servicios de venta en linea; muchas de estas empresas tienden a crecer y requerir de sistemas mas especializados que se ajusten a necesidades particulares.

Se desarrolla un sistema que tiene como objetivo el llevar control de los procesos de venta y logística de una empresa. El propósito del proyecto es integrar los servicios existentes de venta en linea de la compañía y extender su uso, adaptándolos a los requerimientos específicos del cliente, así como diseñar una interfáz gráfica que permita proporcionar un servicio mas rápido y agilice los proceso de elaboración y entrega de productos.

DEDICATORIA

A mis abuelos, por creer siempre en mí
Gracias

AGRADECIMIENTOS

Quiero dar gracias a Bolt Media por incluirme en su equipo y darme la oportunidad de crecer.

Índice general

Índice de Tablas	IX
Índice de Figuras	XI
1. Introducción	1
1.1. Antecedentes y definición del problema	1
1.2. Motivación para atender el problema	2
1.3. Propuesta de solución	2
1.4. Objetivos generales	2
1.5. Objetivos específicos	3
2. Marco teórico	4
2.1. MEAN/MERN	5
2.1.1. Componentes MEAN	5
2.1.2. Base de datos NOSQL	6
2.1.3. Orientado a documentos	6
2.1.4. Angular/React	7

2.1.5.	Beneficios	8
2.1.6.	Desventajas	8
2.2.	FERN	9
2.2.1.	Componentes	9
2.2.2.	Cloud Firestore	10
2.2.3.	Node.js	11
2.2.4.	El ecosistema NPM	12
2.2.5.	Comandos NPM	13
2.2.6.	Express.js	14
2.2.7.	React.js	15
2.2.8.	Componentes React	15
2.2.9.	DOM Virtual	18
2.3.	Preprocesamiento	19
2.3.1.	ES5/ES6	19
2.4.	Control de versiones	22
2.4.1.	Git	22
3.	Análisis y diseño del sistema	24
3.1.	E-Commerce (Comercio electrónico)	24
3.2.	Requerimientos del proyecto	26
3.2.1.	Herramientas	27
3.3.	Configuración del servidor (Backend)	32
3.3.1.	Servidor NodeJS	32

3.3.2.	Configuración Express	35
3.3.3.	Configuración Firebase	37
3.3.4.	Autenticación	42
3.3.5.	Integración con Shopify	44
3.3.6.	Webhooks	45
3.3.7.	Webhooks Shopify	45
3.3.8.	Configuración de webhook en Express.js	47
3.3.9.	Websockets con Socket.IO	48
3.4.	Desarrollo de la aplicación web (Frontend)	50
3.4.1.	Patrones de diseño	50
3.4.2.	Redux	50
3.5.	Integración React/Redux	53

Índice de tablas

Índice de figuras

2.1. Flujo de informacion en MEAN Stack.	6
2.2. Diagrama que representa las diferencias clave entre la base de datos SQL y las bases de datos NoSQL.	7
2.3. Flujo de informacion en FERN Stack.	10
2.4. Colección de Firestore con sus documentos internos.	11
2.5. Analogía Node.js con Java.	12
2.6. Componentes principales de una página web.	16
2.7. Diferencias entre controladores de versiones.	23
3.1. Tienda en linea actual de la empresa.	25
3.2. Interfaz del administrador Shopify.	26
3.3. Vista general de la raíz del proyecto.	30
3.4. Comparación de node y servidores tradicionales.	33
3.5. Cada función de middleware en la pila se ejecuta antes que las que están debajo de ella.	35
3.6. Consola de Google Firebase.	37
3.7. Autenticación basada en roles.	42
3.8. Panel para crear webhooks en Shopify.	46

3.9. Comparación de transporte por sondeo (Ajax) y websockets (Socket.IO).	48
3.10. Comparación Flux/Redux.	51

Código Ejemplo

2.1. Simple aplicación Express.js	14
2.2. Ejemplo de página con componentes React.js	17
2.3. Ejemplo de sintaxis ES5 y ES6	20
3.1. Variables principales del sistema	31
3.2. Configuración servidor NodeJS básico	34
3.3. Fragmento de la configuración de enrutamiento del sistema . .	36
3.4. Fragmento de la configuración Firebase en el servidor	40
3.5. Fragmento para guardar datos en Firestore	41
3.6. Fragmento para guardar crear y leer Firebase Custom Claims	43
3.7. Fragmento de código para la conexión con Shopify	45
3.8. Ruta que capta el webhook lanzado desde Shopify	47
3.9. Fragmento de configuración de Socket.IO del sistema	49
3.10. Fragmento de código para inicializar el Redux Store	51
3.11. Fragmento de código del reducer común de la app	52
3.12. Fragmento de una acción que actualiza el estado	52
3.13. Fragmento de código de la aplicación React principal	53

Capítulo 1

Introducción

Muchas pequeñas empresas dependen en gran medida al éxito de sus ventas en línea, por lo que adoptan el uso de tecnologías robustas (como Shopify y WooCommerce) para su administración. su función principal es servir como puntos de entrada para los clientes pero dichos servicios no están diseñados para mostrar información detallada a distintas áreas de la empresa.

1.1. Antecedentes y definición del problema

Se solicitó a la empresa Bolt Media Internacional S. de R.L. de C.V. desarrollar una plataforma que conectara las transacciones de rosesland.com, una plataforma de ventas en línea desarrollada en Shopify a las operaciones internas de la florería, como lo son las ventas de mostrador, la manufactura y la entrega de los productos.

Actualmente la florería realiza todo este proceso con comandas que los empleados de ventas llenan a mano, para después enviarlos al área de elaboración de arreglos florales y posteriormente se entregan al encargado de logística para dar indicaciones de la distribución de los productos. Este proceso de venta es lento y genera una experiencia poco placentera para los compradores, a su vez el área de manufactura está conformado por artesanos y trabajadores del campo que tienen un nivel de comprensión de lectura bajo y pueden llegar a

cometer errores que retrasen todos los procesos. Por tales motivos este sistema debe ser fácil de usar para usuarios de distintas áreas y a su vez el diseño debe de adaptarse tanto a diferentes tamaños de pantallas como a diferentes dispositivos móviles. Se requiere una base de datos que notifique en tiempo real los cambios en la información así como un servidor que administre los permisos adecuados para su manipulación.

1.2. Motivación para atender el problema

Una de las herramientas mas poderosas para el comercio es el internet, saber aprovecharlo genera importantes beneficios para cualquier negocio, es por ello que la empresa Rosesland debe adoptar un proceso mas automatizado en sus funciones y tomar ventaja de los ordenadores y dispositivos con los que cuenta la empresa.

1.3. Propuesta de solución

Implementar un sistema diseñado principalmente para ser utilizado en pantallas táctiles que muestre las transacciones realizadas durante el día tanto en mostrador como en la pagina web, dando un informe detallado de las operaciones de la compañía.

1.4. Objetivos generales

Automatizar el proceso de ventas y entregas de la floreía Rosesland e integrar los servicios de su tienda en linea rosesland.com mediante una aplicación web progresiva, que sea compatible con los dispositivos existentes de la empresa y que funcione adecuadamente en navegadores de internet modernos.

1.5. Objetivos específicos

1. Desarrollar un servidor que administre la autenticación de los empleados y su acceso a la información.
2. Implementar una base de datos que notifique a los usuarios cambios en la información en tiempo real.
3. Diseñar una aplicación web progresiva que garantice una experiencia de usuario óptima para todas las áreas de la empresa
4. Integrar los servicios de la tienda en linea rosesland.com con las operaciones internas de la empresa, uniendo los procesos en una sola plataforma

Capítulo 2

Marco teórico

Un *stack de aplicaciones* es una colección de software o tecnologías que se utilizan para crear una aplicación web. Las aplicaciones de una sola página (SPA - Single Page Application) han crecido en popularidad ya que proporcionan una experiencia de usuario más fluida: llamadas de servidor livianas cambian lo que se muestra en la pantalla sin tener que actualizar toda la página. El resultado parece bastante ingenioso en comparación con la antigua forma de volver a cargar la página por completo. Esto provocó un aumento en los *frameworks* de *front-end*, ya que gran parte del trabajo se realizó en el lado del cliente. Aproximadamente al mismo tiempo, aunque completamente sin relación, las bases de datos NoSQL también comenzaron a ganar popularidad. El término *stack* fue popularizado por primera vez por LAMP Stack: Linux, Apache, MySQL y PHP. Linux es el sistema operativo, Apache actúa como el servidor HTTP, MySQL proporciona la base de datos relacional para manejar la información de la aplicación y PHP es el lenguaje de programación en el que se construye la aplicación. MERN es un paquete de software que significa MongoDB, Express.js, ReactJS y NodeJS. Juntos, estos programas gratuitos mejoran la simplicidad del proceso de desarrollo web. Las opciones son muchas, pero elegir una puede ser difícil.

2.1. MEAN/MERN

En comparación con LAMP, el paquete de aplicaciones MEAN es bastante nuevo. Una de sus mayores diferencias es que MEAN no depende de un sistema operativo específico. Node.js se encarga de la ejecución del lado del servidor. MEAN Stack se recomienda especialmente para desarrolladores de JavaScript, ya que utiliza JavaScript en todos los niveles, tanto para el código del lado del cliente, así como el código del lado del servidor. Angular, el potente *framework* de *front-end*, utiliza un patrón de diseño Modelo-Vista-Controlador. A medida que React crece en popularidad, ha presentado una opción alternativa para el desarrollo *frontend*, aunque React es simplemente una biblioteca, no un *framework* MVC completo. MongoDB es un juego un papel importante y una opción increíblemente popular en el mundo de la gestión de bases de datos NoSQL. Node.js le permite al programador escribir el back-end de la aplicación en Javascript, Express.js se usa encima de Node para manejar las solicitudes de enrutamiento y proporcionar una API REST, o incluso puede usarse para generar el HTML final para ser utilizado por el *framework* de *front-end*.

2.1.1. Componentes MEAN

- MongoDB (base de datos)
- Express.js (servidor)
- Angular.js (cliente)
- Node.js (entorno del servidor)

Derivados:

- MERN (React.js en lugar de Angular.js)
- MEEN (Ember.js en lugar de Angular.js)

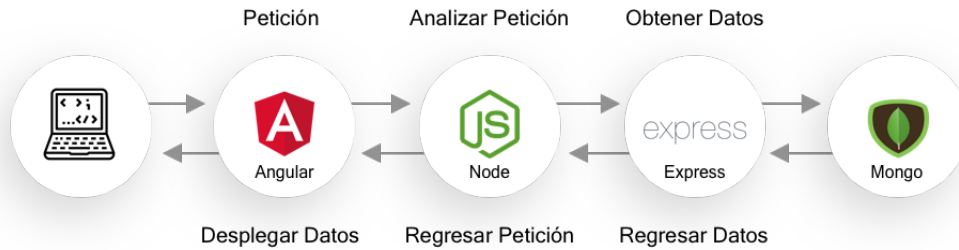


Figura 2.1: Flujo de informacion en MEAN Stack.

2.1.2. Base de datos NOSQL

Las bases de datos NOSQL son una alternativa emergente a las bases de datos relacionales más utilizadas. Como su nombre lo indica, no reemplaza completamente a SQL, sino que lo complementa de tal manera que puedan coexistir

El concepto de NOSQL se desarrolló hace mucho tiempo, pero fue después de la introducción de la base de datos como servicio (DBaaS) que obtuvo un reconocimiento destacado. Debido a la alta escalabilidad proporcionada por NOSQL, fue visto como un importante competidor del modelo de base de datos relacional. A diferencia de RDBMS, las bases de datos NOSQL están diseñadas para escalar fácilmente a medida que crecen. La mayoría de los sistemas NOSQL han eliminado el soporte multiplataforma y algunas características adicionales innecesarias de RDBMS, haciéndolos mucho más livianos y eficientes que sus contrapartes RDMS.

2.1.3. Orientado a documentos

El concepto principal de una base de datos orientada a documentos es que el documento contiene grandes cantidades de datos que pueden estar disponibles de manera útil. Se puede acceder a estos documentos como un directorio regular en el que puede tener diferentes colecciones y cada colección tiene documentos que contienen la información deseada. Además, cada colección puede tener colecciones internas. Puede tener un árbol completo de documentos, sin

embargo, esta práctica no se recomienda y debe evitar tener más de tres niveles de anidamiento.

Las bases de datos NoSQL no son necesariamente bases de datos relacionales. Los datos no son representados en términos de filas y columnas de tablas. En MongoDB, los datos se visualizan como objetos o documentos. Esto ayuda a un programador a evitar una capa de traducción, por lo que no es necesario convertir o asignar los objetos con los que trata el código en tablas relacionales. Dichas traducciones se denominan capas de Mapeo Relacional de Objetos (ORM).

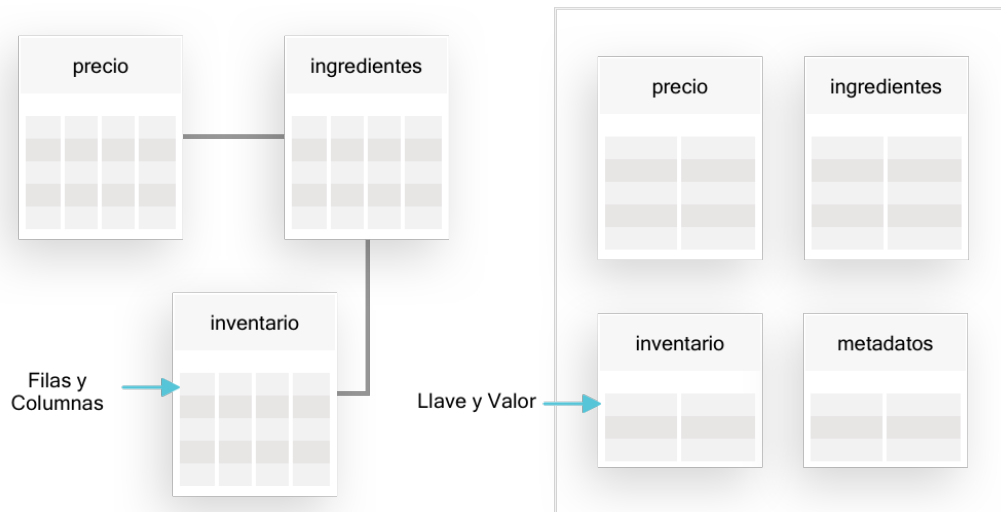


Figura 2.2: Diagrama que representa las diferencias clave entre la base de datos SQL y las bases de datos NoSQL.

2.1.4. Angular/React

Angular o React, proporcionan la interfaz de usuario reactiva de su aplicación. Utilizan componentes, son reactivos porque el usuario recibe cambios inmediatos cuando interactúa con la aplicación y, por lo general, se ejecutan dentro del navegador de un usuario (aunque ambos son isomórficos, capaces de ejecutarse en un servidor).

2.1.5. Beneficios

Usar JavaScript como el lenguaje de programación principal es una gran ventaja. Todo se puede configurar rápidamente y hacer en JavaScript, lo que hace que sea mucho más fácil encontrar desarrolladores, y los desarrolladores de LAMP generalmente también conocen JavaScript. Otra gran ventaja es la capacidad de crear fácilmente aplicaciones móviles o de escritorio, por ejemplo con Ionic. El código y los componentes se pueden reutilizar o agregar fácilmente.

2.1.6. Desventajas

Muchas librerías y *frameworks* son bastante nuevos, y las nuevas versiones se lanzan rápidamente, por lo que mantener una aplicación puede ser una molestia. Dado que muchas tecnologías desaparecen después de unos años, la sostenibilidad puede convertirse en un problema. También es más difícil mantener una base de código limpia y seguir las mejores prácticas a medida que su aplicación crece. Además, debe confiar en el cliente y las tecnologías disponibles del cliente.

2.2. FERN

Firebase es una plataforma propiedad de Google que tiene como objetivo proporcionar un enfoque holístico para un rápido desarrollo web y móvil. En resumen, le permite centrarse en las partes frontales de la aplicación. Se completa con una base de datos visual sin tablas (NoSQL), alojamiento, almacenamiento de archivos, procesamiento del lado del servidor para cosas que deben protegerse de la interfaz y un sistema de autenticación, todo lo necesario para aplicaciones pequeñas y medianas.

Cloud Firestore es el servicio de base de datos de Google Firebase para aplicaciones móviles. Firestore permite una experiencia de programación increíble cuando se usa en un stack de aplicaciones completo. Los datos en tiempo real y la facilidad de conexión a la base de datos hacen que FERN Stack sea una forma rápida de conectar estas tecnologías.

2.2.1. Componentes

- Firebase (base de datos)
- Express.js (servidor)
- React.js (cliente)
- Node.js (entorno del servidor)

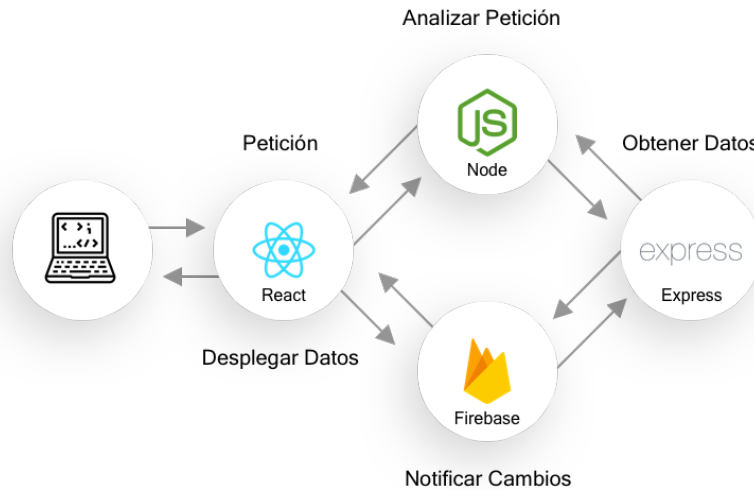


Figura 2.3: Flujo de informacion en FERN Stack.

2.2.2. Cloud Firestore

Firestore es una bases de datos orientada a documentos, toda la información se guarda en colecciones como JSON, principalmente diseñada para almacenar, recuperar y administrar información orientada a documentos, también conocida como datos semiestructurados.

La escalabilidad es completamente automática, lo que significa que no es necesario compartir sus datos en varias instancias. Los cargos de Cloud Firestore se basan en las operaciones realizadas en su base de datos (lectura, escritura, borrado), ancho de banda y almacenamiento. Admite límites de gasto diario para proyectos de Google App Engine, para garantizar que no exceda los costos con los que el usuario se sienta cómodo.

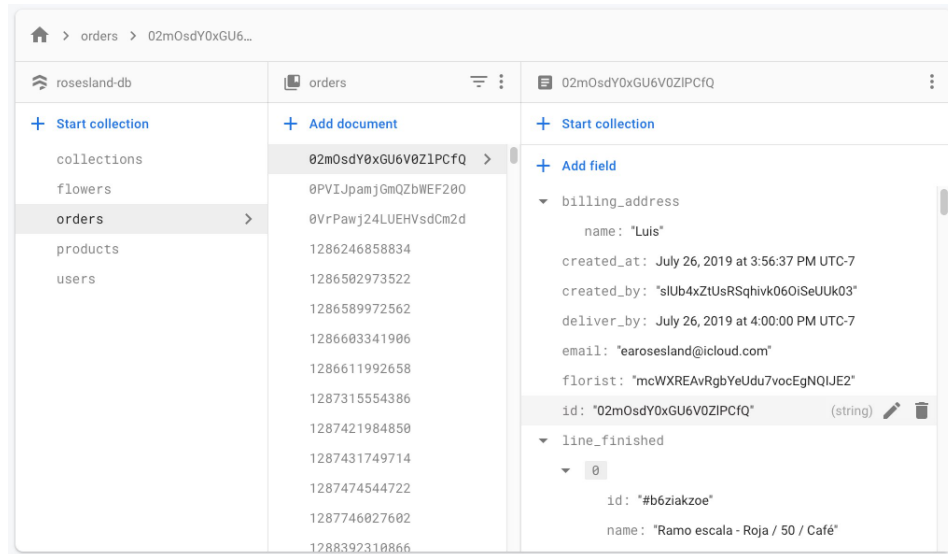


Figura 2.4: Colección de Firestore con sus documentos internos.

2.2.3. Node.js

Node.js es un entorno multiplataforma de código abierto para ejecutar código JavaScript del lado del servidor. El entorno de tiempo de ejecución de Node.js incluye todo lo que necesita para ejecutar un programa escrito en JavaScript.

Node.js surgió cuando los desarrolladores originales de JavaScript lo extendieron de algo que solo podía ejecutar en el navegador a algo que podría ejecutar en su máquina como una aplicación independiente. Ahora puede hacer mucho más con JavaScript que simplemente hacer que los sitios web sean interactivos. JavaScript ahora tiene la capacidad de hacer cosas que otros lenguajes de secuencias de comandos como Python pueden hacer. Tanto su navegador JavaScript como Node.js se ejecutan en el motor de tiempo de ejecución JavaScript V8. Este motor toma su código JavaScript y lo convierte en un código de máquina más rápido. El código de máquina es un código de bajo nivel que la computadora puede ejecutar sin necesidad de interpretarlo primero.

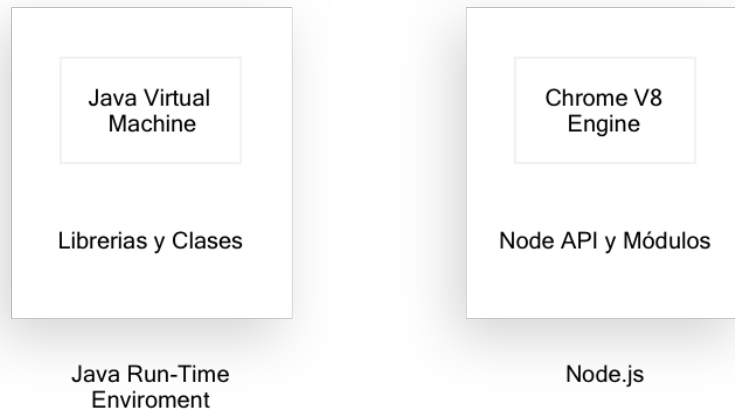


Figura 2.5: Analogía Node.js con Java.

Motor V8 de Google Chrome

Node.js utiliza el motor de ejecución ultra rápido V8 de Google Chrome. Hasta el lanzamiento de Chrome, la mayoría de los navegadores leían JavaScript de manera ineficiente: el código se leía e interpretaba poco a poco. Tomó mucho tiempo leer JavaScript y convertirlo a lenguaje máquina para que el procesador pudiera entenderlo.

El motor V8 de Google Chrome funciona completamente diferente. Está altamente optimizado y lleva a cabo lo que llamamos compilación JIT (Just In Time). Transforma rápidamente el código JavaScript en lenguaje máquina.

2.2.4. El ecosistema NPM

NPM (Node Package Manager) es el administrador de paquetes predeterminado para Node.js. se instala en el sistema con la instalación de Node.js. Los paquetes y módulos necesarios en un proyecto Node se instalan utilizando *npm*.

NPM consta de tres componentes:

1. Sitio web
2. Registro
3. CLI

Sitio web

El sitio web oficial de npm es <https://www.npmjs.com/>. Con este sitio web puede encontrar paquetes, ver documentación, compartir y publicar paquetes.

Registro

El registro npm es una gran base de datos que consta de más de medio millón de paquetes. Los desarrolladores descargan paquetes del registro npm y publican sus paquetes en el registro.

CLI (interfaz de línea de comando)

Esta es la línea de comando que ayuda a interactuar con el npm para instalar, actualizar y desinstalar paquetes y administrar dependencias.

2.2.5. Comandos NPM

Npm tiene muchos paquetes que puedes usar en una aplicación para que su desarrollo sea más rápido y eficiente. Instalar módulos usando NPM no representa un gran problema. Hay una sintaxis simple para instalar cualquier módulo Node.js:

```
npm install nombre-del-paquete  
ejemplo: npm install express
```

2.2.6. Express.js

Escribir un servidor web completo a mano en Node.js directamente no es tan fácil, ni es necesario, Express.js es un paquete de aplicación web minimalista y extensible creado para el ecosistema Node.js. Permite crear un servidor web legible, flexible y fácil de mantener.

Express.js le permite definir rutas, especificaciones de qué hacer cuando llega una solicitud HTTP que coincide con un patrón determinado. La especificación coincidente se basa en expresiones regulares (regex) y es muy flexible, como la mayoría de los otros entornos de aplicaciones web. La parte de qué hacer es solo una función que recibe la solicitud HTTP analizada.

Express.js analiza la URL de solicitud, encabezados y parámetros. En el lado de la respuesta, tiene, como se esperaba, toda la funcionalidad requerida por las aplicaciones web. Esto incluye la configuración de códigos de respuesta, configuración de cookies, envío de encabezados personalizados, etc. Además, puede escribir middleware Express, piezas de código personalizadas que se pueden insertar en cualquier ruta de procesamiento de solicitud/respuesta para lograr una funcionalidad común como el registro, la autenticación, entre otras.

Código Ejemplo 2.1: Simple aplicación Express.js

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => res.send('Hola mundo!'));

app.listen(port, () => console.log(`Servidor iniciado el puerto ${port}`));
```

2.2.7. React.js

React es una biblioteca de JavaScript declarativa, eficiente y flexible creada en 2013 por el equipo de desarrollo de Facebook. React quería que las interfaces de usuario fueran más modulares (o reutilizables) y más fáciles de mantener. Según el sitio web de React, se utiliza para *construir componentes encapsulados que administran su propio estado, y unirlos para crear interfaces de usuario complejas*. React es una biblioteca JavaScript que permite componer interfaces de usuario complejas a partir de piezas de código pequeñas y aisladas llamadas *componentes*.

En terminos generales, al crear aplicaciones con React.js, se crean componentes que corresponden a distintos elementos de una interfaz de usuario. Después se organizan estos elementos dentro de componentes de orden superior que definen la estructura de la aplicación. Es importante destacar que cada componente en una aplicación React se rige por principios estrictos de gestión de datos. Interfaces avanzadas comunmente involucran datos complejos y manejo de estado. React.js es limitado y tiene como objetivo darnos las herramientas para poder anticipar cómo se verá una aplicación con un conjunto de circunstancias dado.

2.2.8. Componentes React

Un componente es una pequeña parte de la interfaz de usuario. Todas las piezas reutilizables de una página web se abstraen en un componente.



Figura 2.6: Componentes principales de una página web.

En primer lugar, hay un componente principal llamado componente APP. Este componente de la aplicación contiene cuatro componentes secundarios o se divide en cuatro componentes:

1. Encabezado
2. Barra lateral
3. Contenido
4. Pie de página

La función de cada componente se manejará independientemente con otros componentes. Cada componente es una pieza reutilizable, y se puede pensar en cada componente de forma aislada.

Dentro de un componente, tendremos subcomponentes o componentes dentro de un componente padre. Esos serán reutilizables también.

Código Ejemplo 2.2: Ejemplo de página con componentes React.js

```
class Header extends React.Component {
  render() {
    return (
      <header className="navigation">
        {this.props.name}
      </header>
    );
  }
}

class Content extends React.Component {
  render() {
    return (
      <div className="content">
        Hola Mundo
      </div>
    );
  }
}

class App extends React.Component {
  render() {
    return (
      <div>
        <Header name="Mi App" />
        <Content />
      </div>
    );
  }
}
```

2.2.9. DOM Virtual

React utiliza un DOM virtual, que es una representación virtual del DOM (Document Object Model). Detrás de escena, React hace un gran trabajo para editar y volver a renderizar eficientemente el DOM cuando algo en la interfaz necesita cambiar.

2.3. Preprocesamiento

2.3.1. ES5/ES6

ES5 (ES significa ECMAScript) es básicamente ‘JavaScript normal’. La quinta actualización de JavaScript, ES5 se finalizó en 2009. Ha sido compatible con todos los principales navegadores durante muchos años.

ES6 es una nueva versión de JavaScript que agrega algunas buenas adiciones sintácticas y funcionales. Se finalizó en 2015. ES6 es casi totalmente compatible con todos los navegadores principales. Pero pasará algún tiempo hasta que las versiones anteriores de los navegadores web estén fuera de uso. Por ejemplo, Internet Explorer 11 no es compatible con ES6, pero tiene aproximadamente el 8 % de uso de mercado de navegadores.

Para aprovechar los beneficios de ES6 hoy, se tienen que hacer que hacer algunos procedimientos para que funcione en tantos exploradores de internet como sea posible:

1. Se debe preprocesar el código para que una gama más amplia de navegadores entiendan nuestro JavaScript. Esto significa convertir ES6 JavaScript en ES5 JavaScript.
2. Tenemos que incluir un ‘shim’ o ‘polyfill’ que brinde una funcionalidad adicional agregada en ES6 que un navegador puede o no tener.

JSX

JSX es una extensión de sintaxis similar a XML para ECMAScript sin ninguna semántica definida. NO está destinado a ser implementado por motores o navegadores. NO es una propuesta para incorporar JSX en la propia especificación ECMAScript. Está destinado a ser utilizado por varios preprocesadores (transpiladores) para transformar estos tokens en ECMAScript estándar.

BabelJS

BabelJS es un transpilador de JavaScript que transpila nuevas características ES6 al antiguo estándar ES5. Con esto, las funciones se pueden ejecutar en navegadores antiguos y nuevos, sin problemas.

El preprocesador BabelJS convierte la sintaxis de JavaScript moderno en un formulario, que los navegadores más antiguos pueden entender fácilmente. Por ejemplo, `const` y `let` se convertirán en `var`, la función flecha se convierte en una función normal manteniendo la funcionalidad igual en ambos casos.

Código Ejemplo 2.3: Ejemplo de sintaxis ES5 y ES6

Funciones

```
// ES5
var cuadrado = function cuadrado(num) {
  return num * num;
};
```

```
// ES6
const cuadrado = num => num * num;
```

Acceder a los valores de un objeto

```
var obj1 = { a: 1, b: 2 };
```

```
// ES5
var a = obj1.a;
var b = obj1.b;
```

```
// ES6
var { a, b } = obj1;
```


Sass (Syntactically Awesome Style Sheets)

Sass es un preprocesador de CSS, que ayuda a reducir la repetición con CSS y ahorra tiempo. Es un lenguaje de extensión CSS más estable y potente que describe el estilo de una página estructuralmente. Sus principales atributos son:

1. Es un súper conjunto de CSS, lo que significa que contiene todas las características de CSS y es un preprocesador de código abierto, codificado en Ruby.
2. Proporciona algunas características, que se utilizan para crear hojas de estilo que permiten escribir código más eficiente y fácil de mantener.

Webpack

Webpack es un empaquetador de módulos. Webpack toma un archivo de entrada, encuentra todos los archivos de los que depende y genera un archivo que contiene todo el código de una aplicación. Con él es posible importar archivos CSS e imágenes directamente a JavaScript. Se puede compilar CoffeeScript, TypeScript, SASS y LESS. También es capaz de compilar la sintaxis de ES6 en JavaScript amigable para el navegador. En otras palabras, Webpack toma diferentes archivos (como CSS, JS, SASS, JPG, SVG, PNG, etc.) y los combina en paquetes, un paquete separado para cada tipo de archivo.

2.4. Control de versiones

Hoy en día no es prudente empezar un proyecto web sin una estrategia de respaldo. Debido a que los datos son efímeros y se pueden perder fácilmente, por ejemplo, a través de un cambio de código erróneo o un fallo catastrófico de disco, es aconsejable mantener un archivo de todo el trabajo.

Para proyectos de texto y código, la estrategia de respaldo generalmente incluye control de versiones, o seguimiento y administración de revisiones. Dado su papel fundamental, el control de versiones es más efectivo cuando se adapta a los hábitos y objetivos de trabajo del equipo del proyecto.

En su forma más simple, un sistema de control de versiones proporciona un principio básico y un método para almacenar archivos y cambios realizados en ellos. Esto se logra mediante el uso de un repositorio. El repositorio contiene la versión más reciente de cada archivo y el historial de cambios que ha llevado a esa representación. Por lo general, cada cambio incluye información adicional, como el autor y una breve descripción.

2.4.1. Git

Git es un software de control de versiones distribuido particularmente potente, flexible y de bajo costo que hace que el desarrollo colaborativo sea un placer.

La principal diferencia entre Git y cualquier otro versionador es la forma en que Git piensa acerca de sus datos. Conceptualmente, la mayoría de los otros sistemas almacenan información como una lista de cambios basados en archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) piensan en la información que mantienen como un conjunto de archivos y los cambios realizados en cada archivo a lo largo del tiempo. Git no piensa ni almacena sus datos de esta manera. En cambio, Git piensa en sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirma o guarda el estado de su proyecto en Git, básicamente toma una fotografía de

cómo se ven todos sus archivos en ese momento y almacena una referencia a esa instantánea. Para ser eficiente, si los archivos no han cambiado, Git no almacena el archivo nuevamente, solo un enlace al archivo idéntico anterior que ya ha almacenado.

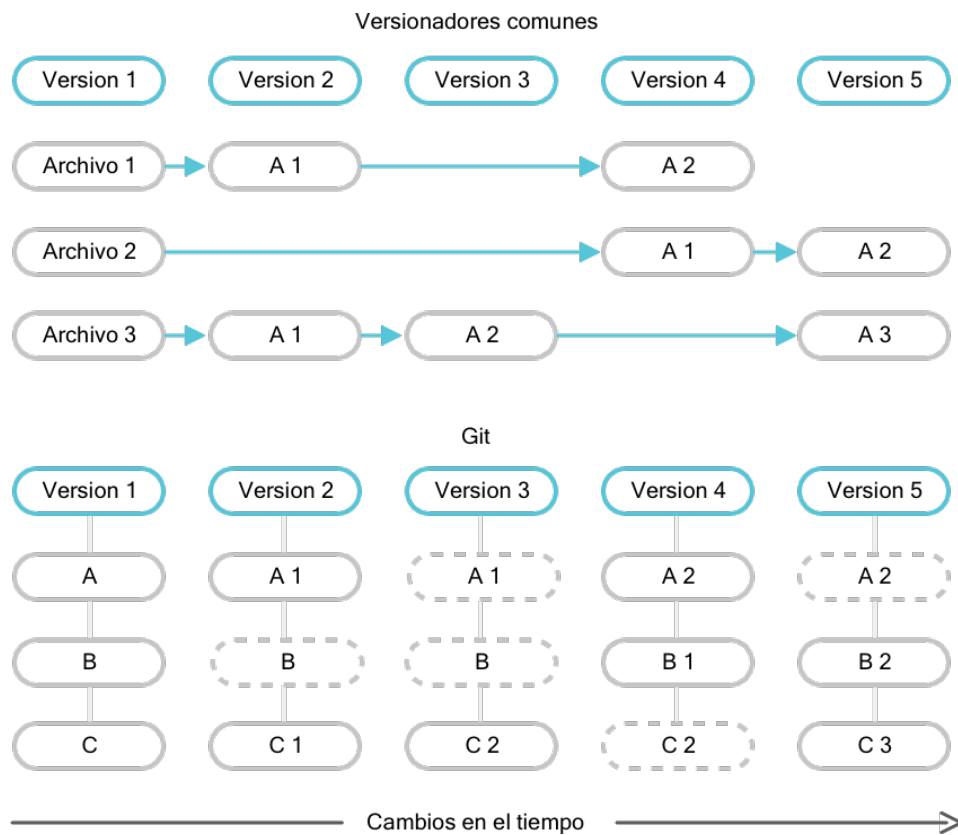


Figura 2.7: Diferencias entre controladores de versiones.

Capítulo 3

Análisis y diseño del sistema

Las microempresas constituyen una potencia impulsora y se consideran entre las principales fuentes de trabajo, no obstante, es muy común encontrar microempresas con dificultades de crecimiento, principalmente debido a una mala administración o por falta de aprovechamiento de sus recursos.

A continuación se describen los procedimientos necesarios para elaborar un sistema que permita a la empresa Rosesland llevar control de las ventas, elaboración y entrega de productos, así como incorporar los servicios de su tienda en línea al sistema. Permitiendo a la empresa tomar ventaja de las tecnologías web mas recientes, aprovechando al máximo los recursos existentes de la compañía.

3.1. E-Commerce (Comercio electrónico)

El comercio electrónico se refiere al proceso de compra o venta de productos o servicios a través de Internet. Las compras en línea se están volviendo cada vez más populares debido a la velocidad y facilidad de uso para los clientes. Las actividades de comercio electrónico, como la venta en línea, pueden dirigirse a consumidores u otras empresas. Vender en línea puede ayudar a su empresa a llegar a nuevos mercados y aumentar sus ventas e ingresos (ya sea a través de su propio sitio web o de un sitio de mercado electrónico).



[Inicio](#) [Catálogos](#) [Especiales](#) [Nosotros](#) [Contacto](#)

[Inicio](#) > [Ramos](#)

Buscar Todo  Ordenar por Más vendidos 

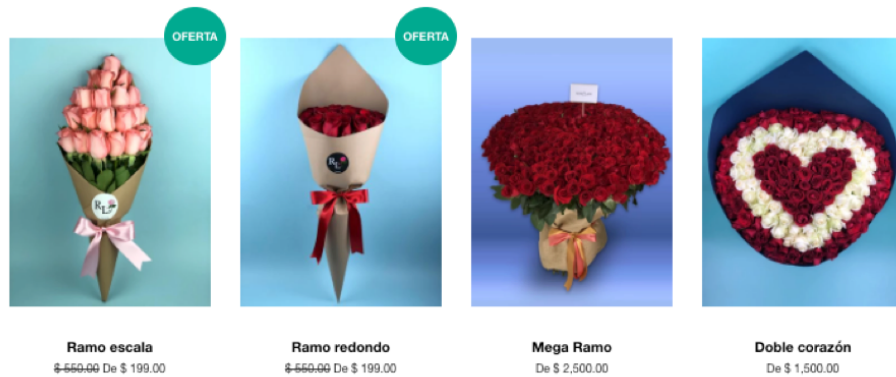


Figura 3.1: Tienda en línea actual de la empresa.

Shopify

Shopify es un servicio web que le permite configurar una tienda en línea para vender sus productos. Le da la facilidad organizar sus productos, personalizar el diseño de su tienda, aceptar pagos con tarjeta de crédito, rastrear y responder a pedidos. Shopify.com permite a los vendedores elegir entre opciones de diseño gratuitas o diseños personalizados creados por los usuarios.

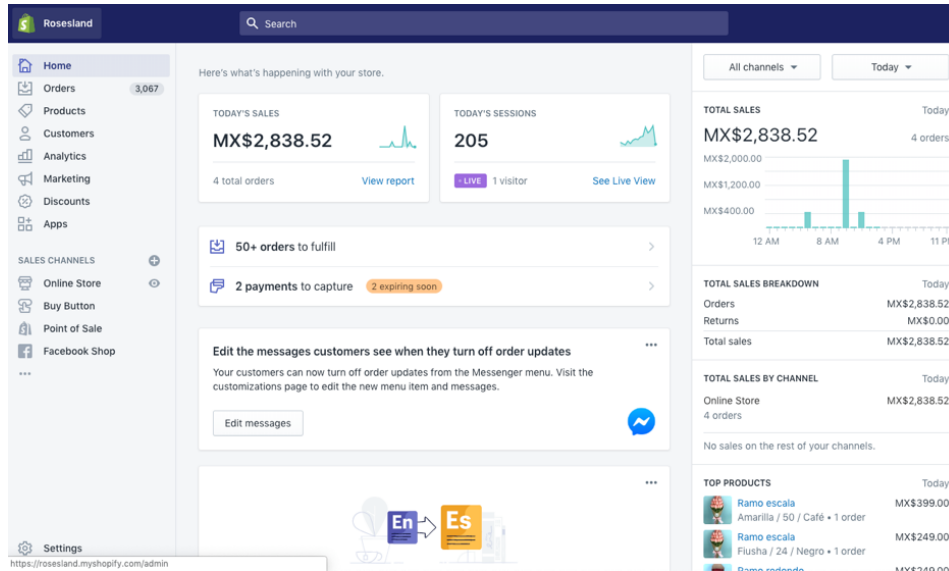


Figura 3.2: Interfaz del administrador Shopify.

3.2. Requerimientos del proyecto

El producto final está enfocado en incrementar las ventas de la empresa Rosesland, con la ayuda de una plataforma que administre y mejore sus procesos internos. La finalidad del proyecto es disminuir los tiempos de ventas, elaboración y entrega de productos, así como la de crear un vínculo entre el servidor y su infraestructura actual extendiendo sus capacidades.

El sistema debe adaptarse al sistema operativo Linux. Anticipado a futuros cambios en la plataforma se decide trabajar con Node.js, ya que es de código abierto y multiplataforma, esto permite beneficiarse de la reutilización del código y la falta de cambio de contexto. Las aplicaciones Node.js están escritas en JavaScript puro y pueden ejecutarse dentro del entorno Node.js en Windows, Linux, etc.

La interfaz gráfica de usuario de la aplicación debe ser responsiva y funcionar en la mayoría de los navegadores web modernos, además debe ser fácil de aprender, idealmente requerir poco entrenamiento.

3.2.1. Herramientas

Node.js permite incorporar herramientas poderosas a proyectos de cualquier tipo. Esto incluye todo, desde bibliotecas y frameworks como jQuery y AngularJS hasta procesadores de código como Webpack. Los paquetes vendrán en una carpeta típicamente llamada `node_modules`, que también contendrá un archivo `package.json`. Este archivo contiene información sobre todos los paquetes, incluidas las dependencias, que son módulos adicionales necesarios para usar un paquete en particular.

Dependencias de desarrollo

Las dependencias de desarrollo son aquellas que se utilizan dentro del entorno de programación. Aquí se incluyen herramientas que no forman parte del ejecutable final como lo son los preprocesadores, empaquetadores y analistas de código. Los principales módulos de desarrollo son los siguientes:

- Babel: es un compilador de JavaScript utilizado para convertir código ECMAScript 2015+ (ES6+) en una versión que pueda ser ejecutado en motores JavaScript más antiguos.
- Webpack: es un empaquetador de módulos principalmente para JavaScript, pero puede transformar archivos front-end como HTML, CSS e imágenes si se incluyen los complementos correspondientes.
- ESLint: es una herramienta de análisis de código estático para identificar patrones problemáticos encontrados en el código JavaScript.
- PostCSS: es una herramienta de desarrollo de software que utiliza complementos basados en JavaScript para automatizar las operaciones de rutina de CSS. Con este módulo es posible analizar CSS, agregar prefijos de proveedor a las reglas CSS, ejecutar optimizaciones enfocadas,

para garantizar que el resultado final sea lo más pequeño posible para un entorno de producción.

- Pug: es un preprocesador que simplifica la tarea de escribir HTML. También agrega funcionalidades, como objetos Javascript, condiciones, bucles, mixins y plantillas.

Estructura del proyecto

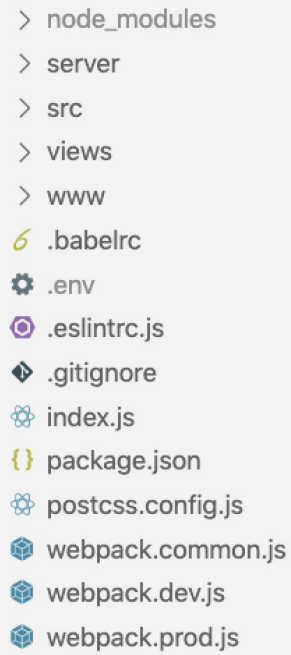
La estructura del proyecto Node.js está influenciada por preferencias personales, la arquitectura del proyecto y la estrategia de inyección de módulos que se está utilizando. Para tener una estructura FERN, es imperativo separar el código fuente del servidor y el utilizado por el cliente, ya que el código del lado del cliente o front-end probablemente se minimizará y se enviará al navegador y es público en su naturaleza básica. Y el lado del servidor o el back-end proporcionarán API para realizar operaciones CRUD.

Directorios

- `node_modules`: Directorio oculto que contiene las dependencias generales del proyecto, este archivo debe ser ignorado por el controlador de versiones.
- `server`: Aquí se encuentran los archivos requeridos por el servidor para el enrutamiento, la conexión con la base de datos y las funciones de utilidad del back-end.
- `src`: Este directorio concentra todos los elementos necesarios para producir nuestra aplicación cliente.
- `views`: En esta carpeta se incluyen los templates necesarios para generar las vistas HTML de nuestro proyecto.
- `www`: Su propósito es contener los archivos del cliente, aquí podemos encontrar el código de producción de nuestra aplicación web compilado y minificado, así como las hojas de estilo, imágenes, fuentes tipográficas, vectores, etc.

Archivos principales

- `.babelrc`: Contiene los mecanismos necesarios para compilar sintaxis moderna de JavaScript y una lista con los navegadores web a los que se requiere dar enfoque.
- `.env`: Documento oculto que contiene las variables del entorno, este archivo es ignorado por el controlador de versiones.
- `.eslintrc.js`: En este archivo se declaran las reglas para identificar e informar sobre patrones o errores encontrados en el código.
- `.gitignore`: Aquí se describen los archivos y directorios que deben ser ignorados por Git.
- `index.js`: Entrada principal de nuestro servidor, contiene el código necesario para que el proyecto funcione correctamente.
- `package.json`: Este archivo puede contener muchos metadatos sobre su proyecto. Pero principalmente se usa para dos cosas:
 - Gestionar dependencias de su proyecto
 - Scripts, que ayudan a generar compilaciones, ejecutar pruebas y otras cosas con respecto al proyecto



- > node_modules
- > server
- > src
- > views
- > www
- .babelrc
- .env
- .eslintrc.js
- .gitignore
- index.js
- package.json
- postcss.config.js
- webpack.common.js
- webpack.dev.js
- webpack.prod.js

Figura 3.3: Vista general de la raíz del proyecto.

Variables del entorno

El acceso a las variables de entorno en Node.js es compatible desde el primer momento. Cuando el proceso Node.js se inicia, automáticamente proporciona acceso a todas las variables de entorno existentes al crear un objeto `env` como propiedad del objeto global del proceso.

Código Ejemplo 3.1: Variables principales del sistema

```
type=""
project_id=""
private_key_id=""
private_key=""
client_email=""
client_id=""
auth_uri=""
token_uri=""
auth_provider_x509_cert_url=""
client_x509_cert_url=""
shopify_secret=""
```

3.3. Configuración del servidor (Backend)

Si alguna vez ha utilizado PHP o ASP, probablemente esté acostumbrado a la idea de que el servidor web (Apache o IIS, por ejemplo) sirve sus archivos estáticos para que un navegador puede verlos a través de la red. Node ofrece un paradigma diferente al de un servidor web tradicional: la aplicación es el servidor web. Node simplemente proporciona las bases para que se pueda construir un servidor web.

3.3.1. Servidor NodeJS

El modelo de E/S impulsado por eventos sin bloqueo le brinda a NodeJS un rendimiento muy atractivo, superando fácilmente los entornos de servidores como PHP y Ruby on Rails, que bloquean las E/S y manejan múltiples usuarios simultáneos en hilos separados para cada uno. Algo importante que se debe saber es que NodeJS no es un framework sino un entorno, hay frameworks que funcionan con Node, como Express y Sails, lo que facilita la creación de aplicaciones.

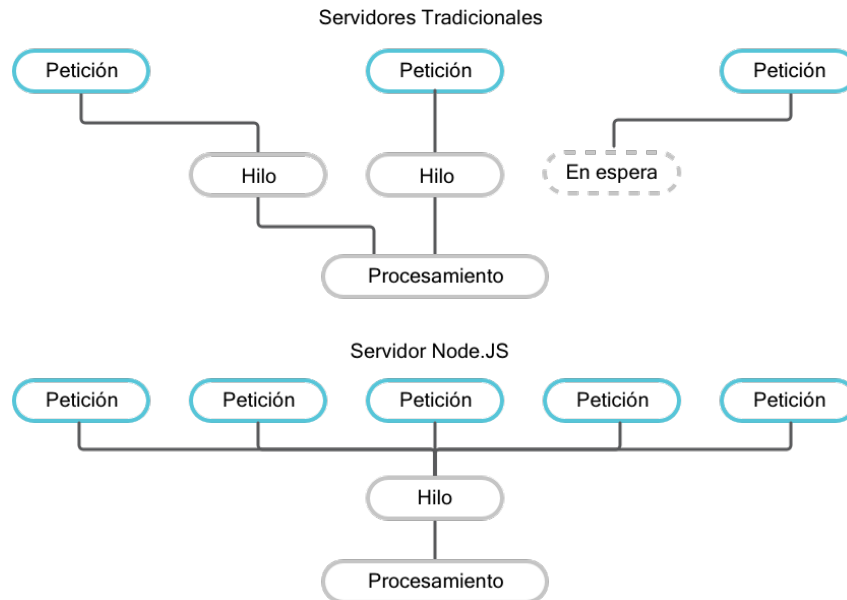


Figura 3.4: Comparación de node y servidores tradicionales.

Un servidor Node.js tiene un solo subproceso de bucle de eventos (event-loop) que espera E/S en sockets y archivos. Una vez que los datos están listos, activa el método de evento correspondiente y espera hasta que regrese antes de esperar nuevamente por más eventos de E/S. Dado que todas las operaciones de E/S no bloquean, se asegurará de que todo se ejecute correctamente tan pronto como la entrada esté disponible sin ningún bloqueo y sin que se tenga lidiar con problemas de subprocesos múltiples.

Programación basada en eventos

La filosofía central detrás de NodeJS es la programación basada en eventos. Significa que, el programador, debe comprender qué eventos están disponibles y cómo responder a ellos. Muchas personas se introducen en la programación basada en eventos mediante la implementación de una interfaz de usuario: el usuario hace clic en algo y se dispara el ‘evento clic’. Es una buena metáfora, porque se entiende que el programador no tiene control sobre cuándo, o si el usuario va a hacer clic en algo, por lo que la programación basada en eventos es realmente bastante intuitiva.

Código Ejemplo 3.2: Configuración servidor NodeJS básico

```
// Carga el módulo http
const http = require('http');
http.createServer((request, response) => {
  // Indica que todo está bien (código 200), y los datos están en texto
  plano
  response.writeHead(200, {
    'Content-Type': 'text/plain'
  });
  // Escribe el texto anunciado en el cuerpo de la página.
  response.write('Hola, Mundo!');
  // Indica al servidor que se han enviado el encabezado y el cuerpo.
  response.end();
}).listen(1337); // Dice al servidor en qué puerto debe escuchar
```

En el ejemplo de código 3.2, el evento es implícito: el evento que se está manejando es una solicitud HTTP. El método `http.createServer` toma una función como argumento; esta función se invocará cada vez que se realice una solicitud HTTP. El programa simplemente establece el tipo de contenido en texto sin formato y envía la cadena ‘Hola, mundo!’.

3.3.2. Configuración Express

Express.js es un ‘marco de aplicación web NodeJS minimalista y flexible’ [4]. Es una capa delgada de características, fundamental para cualquier aplicación web, agrega tres características poderosas: enrutamiento, mejores manejadores de solicitudes y vistas.

Enrutamiento

Enrutamiento se refiere al mecanismo para servir al cliente el contenido que ha solicitado. Para las aplicaciones cliente/servidor basadas en web, el cliente especifica el contenido deseado en la URL (ruta y cadena de consulta).

Cuando una aplicación Express.js se está ejecutando, escucha las solicitudes. Cada solicitud entrante se procesa de acuerdo con una cadena definida de middlewares y rutas que comienzan de arriba a abajo. Este aspecto es importante porque le permite controlar el flujo de ejecución [3].

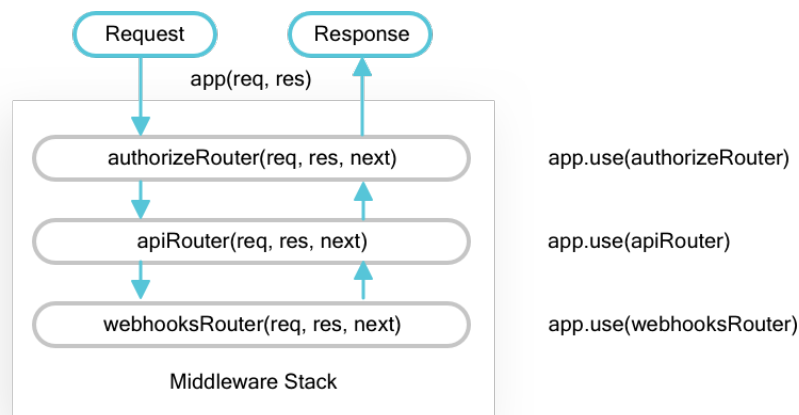


Figura 3.5: Cada función de middleware en la pila se ejecuta antes que las que están debajo de ella.

Código Ejemplo 3.3: Fragmento de la configuración de enrutamiento del sistema

```
const express = require('express');

// Utilidad para rutas de archivos y directorios
const path = require('path');

// Módulos para análisis de solicitudes entrantes
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');

// Inicialización y configuraciones
const app = express();
const http = require('http').Server(app);
app.set('PORT', process.env.PORT || 3000); // Puerto del servidor
require('pug'); //Sistema de plantillas Pug
app.set('view engine', 'pug');
app.set('views', path.join(__dirname, 'views'));

// Middlewares
app.use(cookieParser());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use('/webhooks', (req, res, next) => {...});
app.use('/api', (req, res, next) => {...});

// Enrutamiento
app.get('/', (req, res) => { res.render('index'); });
app.post('/api/user-create', (req, res) => {...});
app.post('/api/user-update', (req, res) => {...});
app.post('/api/order-create', (req, res) => {...});
app.post('/api/order-update', (req, res) => {...});

// Ruta para webhook shopify
app.post('/webhooks/orders/create', async (req, res) => {...});

app.use(express.static('www')); // Carpeta publica de archivos estáticos

http.listen(app.get('PORT'), (error) => {...});
```


3.3.3. Configuración Firebase

Antes de conectar nuestro sistema con una base de datos de Firestore es necesario crear una aplicación en la consola de Google Firebase y seguir los siguientes pasos.

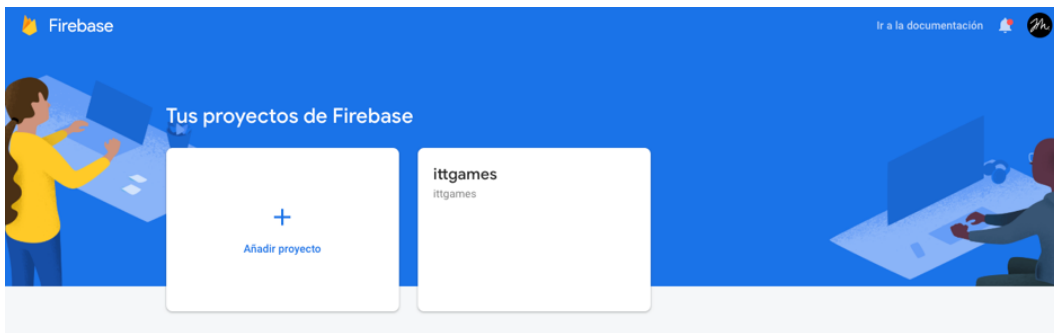
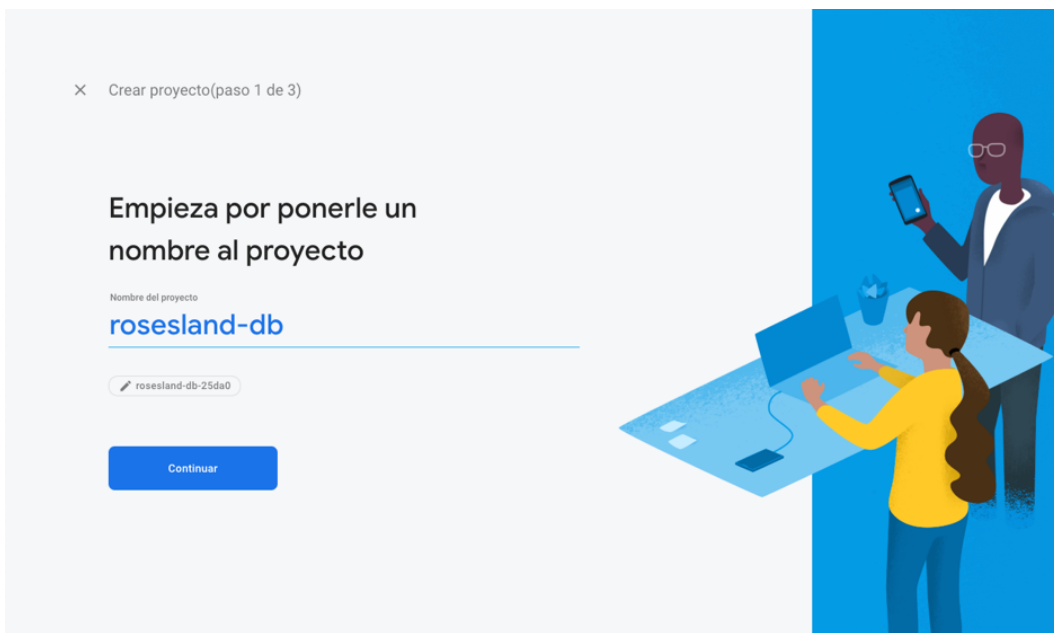
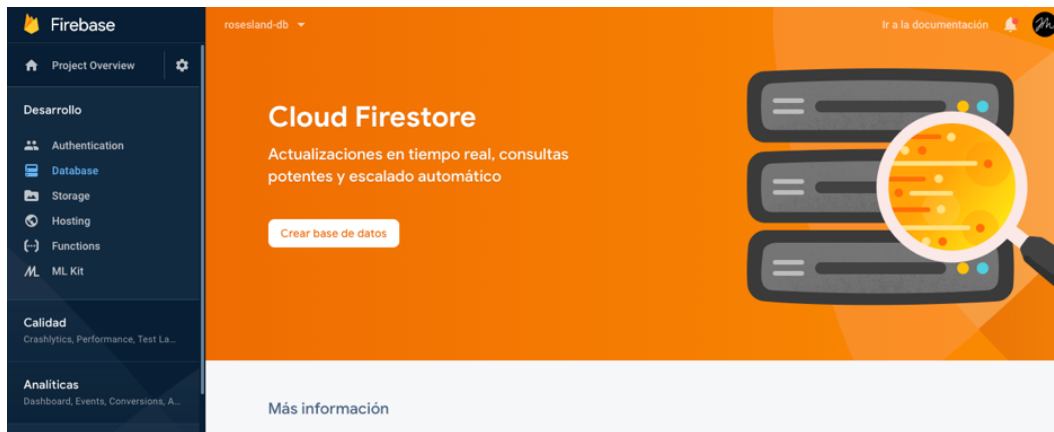


Figura 3.6: Consola de Google Firebase.

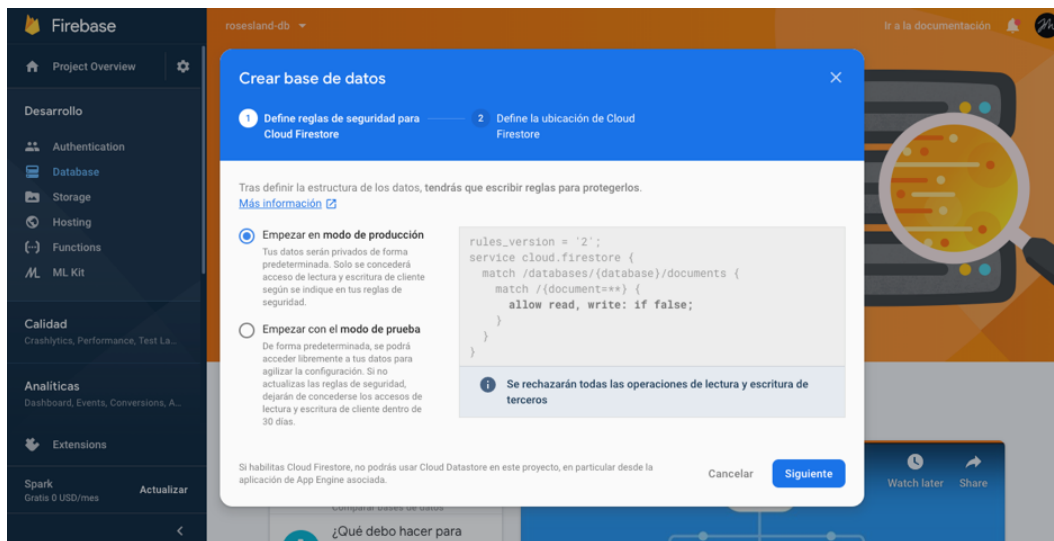
1. Crear un nuevo proyecto en la consola de Firebase.



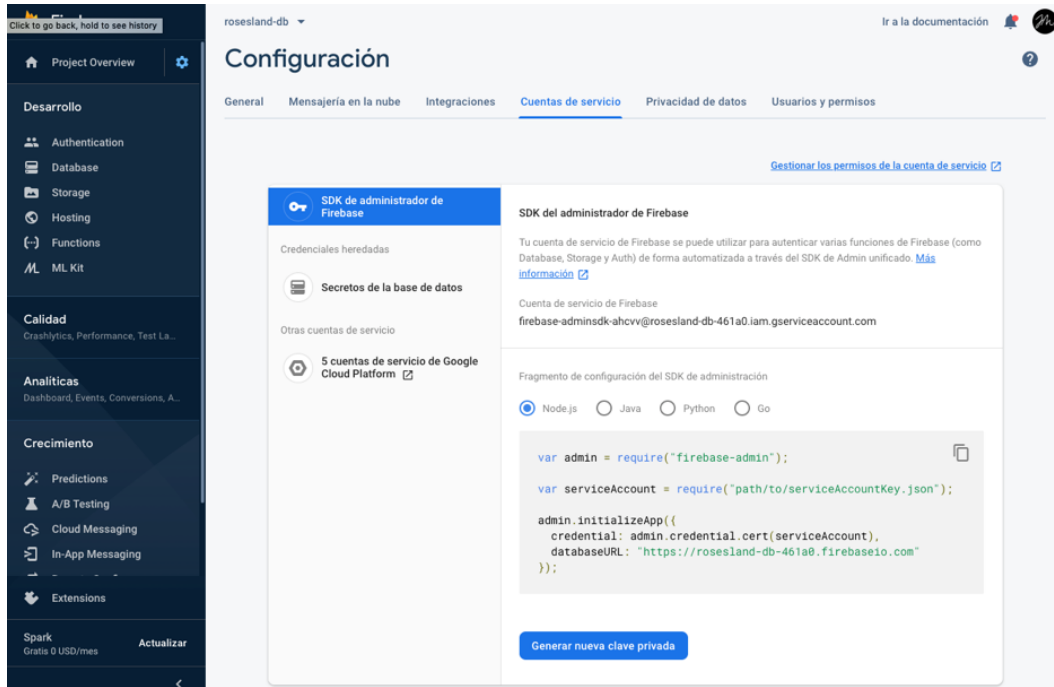
2. En el apartado 'Database', seleccionar 'crear base de datos' .



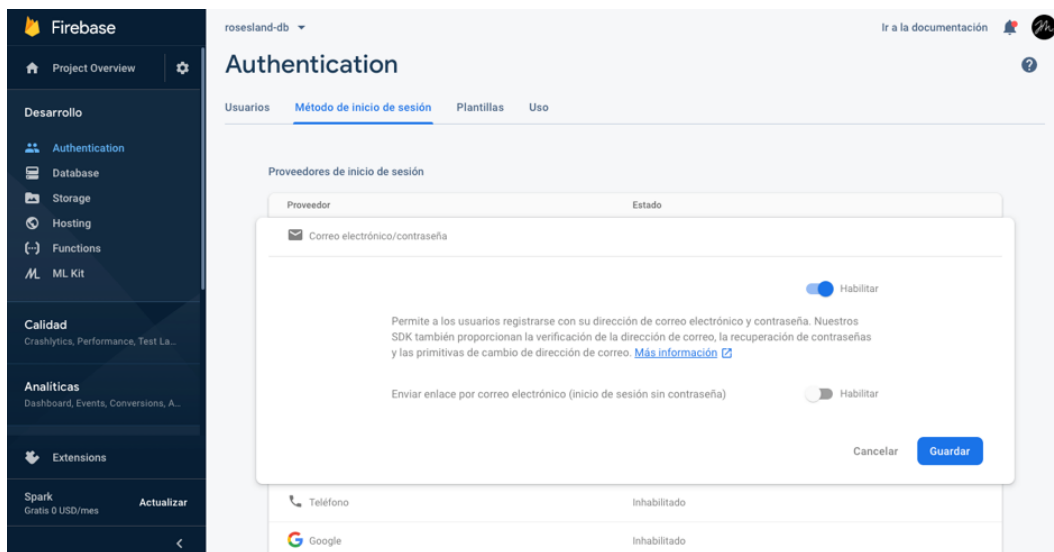
3. En la ventana emergente, seleccionar reglas de seguridad y definir ubicación del servidor.



4. El siguiente paso es generar las claves que permitan usar 'firebase-admin' en el backend. En la configuración del proyecto debajo del apartado 'cuentas de servicio', presionar el botón 'Generar cuentas de servicio'. El contenido del archivo JSON generado se debe de agregar a las variables del entorno que correspondan.



5. Por ultimo en la sección 'Authentication' se deben activar los servicios de autenticación necesarios, para este proyecto es necesario activar la validación por correo electrónico y contraseña.



Configuración de Firebase Firestore en Node.js

Después de crear el objeto de configuración, es necesario inicializar Firebase en la aplicación de la siguiente manera:

Código Ejemplo 3.4: Fragmento de la configuración Firebase en el servidor

```
// Módulo permite el acceso a los servicios de Firebase.
const admin = require('firebase-admin');

// Información de la cuenta de servicio almacenado en las variables del
entorno.
const serviceAccount = {
  type: process.env.type,
  project_id: process.env.project_id,
  private_key_id: process.env.private_key_id,
  private_key: process.env.private_key.replace(/\n/g, '\n'),
  client_email: process.env.client_email,
  client_id: process.env.client_id,
  auth_uri: process.env.auth_uri,
  token_uri: process.env.token_uri,
  auth_provider_x509_cert_url: process.env.auth_provider_x509_cert_url,
  client_x509_cert_url: process.env.client_x509_cert_url,
};

const adminconfig = {
  credential: admin.credential.cert(serviceAccount),
  databaseURL: 'https://rosesland.firebaseio.com',
};

// Inicialización de Firebase.
admin.initializeApp(adminconfig);
```

Escrituras en lotes

Para manipular documentos en un conjunto de operaciones, se pueden ejecutar varias operaciones de escritura como un lote único que incluya cualquier combinación de operaciones `set()`, `update()` o `delete()`. El lote de escrituras se completa de forma atómica y puede escribir en varios documentos. Los siguientes ejemplos muestran cómo crear y confirmar un lote de escrituras [1]:

Código Ejemplo 3.5: Fragmento para guardar datos en Firestore

```
// Inicialización de Firebase.
admin.initializeApp(adminconfig);

// Inicialización de Firebase Firestore.
const db = admin.firestore();

// Referencia a la colección de productos.
const productsRef = db.collection('products');

// Función que guarda productos en la base de datos.
const productsUpdate = (products) => {
  // Generar escritura por lotes.
  const batch = db.batch();
  products.map((product) => {
    const productRef = productsRef.doc(product.id.toString());
    // Agregar la función set() a la pila de transacciones.
    batch.set(productRef, product, { merge: true });
  });
  // Ejecutar lote de transacciones.
  return batch.commit()
    .then(() => {
      console.log('Productos actualizados');
    })
    .catch((error) => {
      console.error('Error', error);
    });
};
```

3.3.4. Autenticación

Casi todas las aplicaciones requieren algún sistema de autorización. En algunos casos, validar un nombre de usuario/contraseña establecido con nuestra tabla de Usuarios es suficiente, pero a menudo, necesitamos un modelo de permisos más detallado para permitir que ciertos usuarios accedan a ciertos recursos y los restrinjan de otros. Construir un sistema para soportar esto último no es trivial y puede llevar mucho tiempo. El API de autenticación basada en roles de Firebase, ayuda a poner todo en marcha rápidamente.

Autenticación basada en roles

En este modelo de autorización, se otorga acceso a roles, en lugar de usuarios específicos, y un usuario puede tener uno o más, según cómo diseñe su modelo de permiso. Los recursos, por otro lado, requieren ciertos roles para permitir que un usuario lo ejecute.

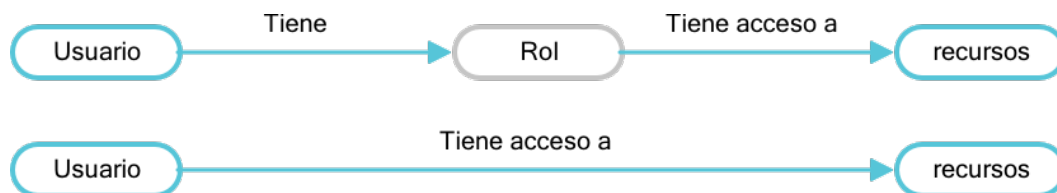


Figura 3.7: Autenticación basada en roles.

Firestore Custom Claims

Los roles de usuario son necesarios para identificar a los usuarios como administradores, gerentes o simplemente como clientes. Firestore Custom Claims permite establecer atributos de usuario simples directamente en el JWT del usuario (por ejemplo: { admin: true }). Un JWT es un Json Web Token, es el objeto que contiene la información del usuario actual.

Código Ejemplo 3.6: Fragmento para guardar crear y leer Firebase Custom Claims

```
// Inicialización de Firebase.
admin.initializeApp(adminconfig);

// Función que agrega permisos a un usuario.
const adminAddClaims = (uid, claims) => (
  admin.auth().setCustomUserClaims(uid, claims)
    .then(() => {
      // Los nuevos Claims se propagarán al token del usuario la próxima
      // vez que se emita uno.
      return { data: uid };
    })
    .catch(error => error)
);

// Función que valida y lee los permisos del usuario.
const verifyIdToken = token => (
  admin.auth().verifyIdToken(token)
    .then(claims => claims)
    .catch(error => error)
);

// Middleware de validación de usuario para todas las entradas al API.
app.use('/api', (req, res, next) => {
  const { token } = req.cookies;
  if (!token) {
    res.status(400).send({ data: 'error' });
  } else {
    verifyIdToken(token)
      .then((claims) => {
        req.claims = claims; // Agrega los permisos a la cadena de petición.
        next();
      })
      .catch((error) => {
        res.status(400).send({ data: 'error', message: error });
      });
  }
});
```

3.3.5. Integración con Shopify

La API de Storefront de Shopify brinda el control creativo completo para crear experiencias de compra personalizadas; se centra en las experiencias de compra vistas desde la perspectiva del cliente. Las características principales del API Storefront son [5]:

- Obtener datos sobre un solo producto o una colección de productos para mostrar en cualquier sitio web o dispositivo.
- Crear experiencias de pago únicas con control total sobre el carrito de compras.
- Crear nuevos clientes o modifique los existentes, incluida la información de la dirección.

Los motivos aquí son bastantes simples. Se requiere que los usuarios puedan navegar, buscar y seleccionar productos directamente en un dominio personalizado sin tener que ir a la tienda en línea de Shopify.

Una vez realizada una conexión exitosa con la API Storefront de Shopify, es posible crear componentes React.js para visualizar imágenes de productos, variaciones de productos, tamaños de productos, etcétera. De este modo el sistema podrá extender el uso de la plataforma.

Código Ejemplo 3.7: Fragmento de código para la conexión con Shopify

```
// Enlaces oficiales del API de Shopify para Node.js.
const Shopify = require('shopify-api-node');

// Iniciar conexión con Shopify.
const shopify = new Shopify({
  shopName: 'rosesland',
  apiKey: '<clave api>',
  password: '<palabra secreta>',
});

const getProducts = () => (
  // Función de lectura de productos de Shopify.
  shopify.product.list()
    .then((products) => {
      // Función que guarda productos en Firestore.
      productsUpdate(products);
    })
    .catch(err => console.error(err))
);
```

3.3.6. Webhooks

Un webhook permite que servicios de terceros envíen actualizaciones en tiempo real a su aplicación. Las actualizaciones se activan por algún evento o acción por parte del proveedor de webhook, y se envían a su aplicación a través de solicitudes HTTP. Cuando recibe la solicitud, la maneja con una lógica personalizada, como enviar un correo electrónico o almacenar los datos en una base de datos.

3.3.7. Webhooks Shopify

Un webhook se puede usar para recibir notificaciones sobre eventos particulares en una tienda en línea. Después de suscribirse a un webhook, puede

permitir que su aplicación ejecute código inmediatamente después de que ocurran eventos específicos en las tiendas que tienen su aplicación instalada, en lugar de tener que hacer llamadas API periódicamente para verificar su estado. Por ejemplo, puede configurar un webhook para activar una acción en su aplicación cuando un cliente crea un carrito de compras o cuando un comerciante cree un nuevo producto en su administrador de Shopify. Al usar las suscripciones de webhooks, puede hacer menos llamadas API en general, lo que garantiza que la aplicación sea más eficiente y se actualice rápidamente [6].

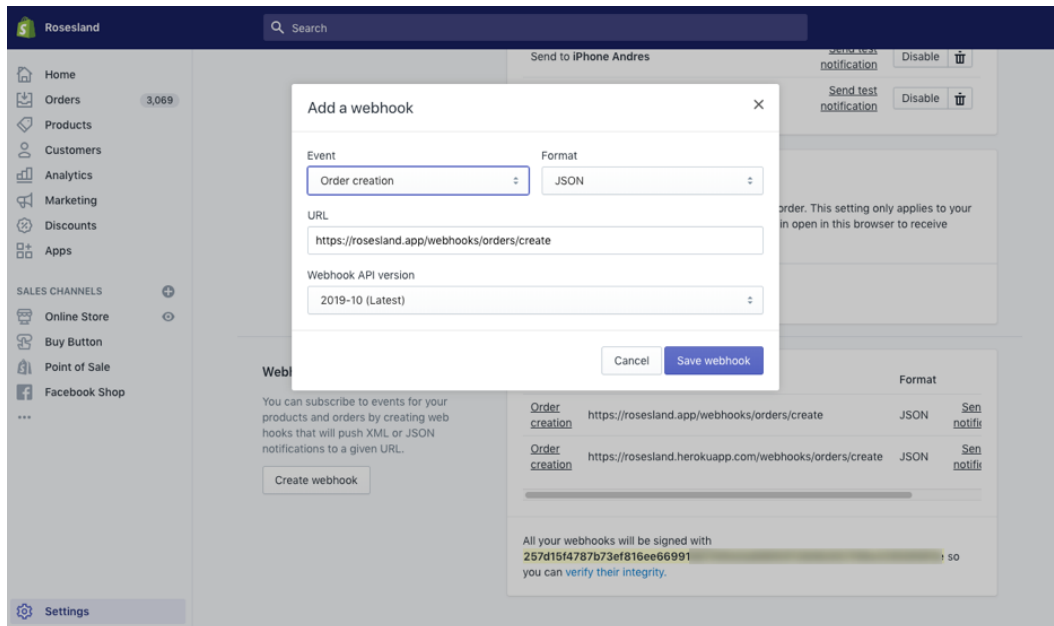


Figura 3.8: Panel para crear webhooks en Shopify.

Después de configurar una suscripción de webhook, los eventos que especificó activarán una notificación de webhook cada vez que ocurran. Esta notificación contiene información JSON y encabezados HTTP que proporcionan contexto.

3.3.8. Configuración de webhook en Express.js

En lugar de extraer información a través del API, los webhooks enviarán información a su punto final.

Código Ejemplo 3.8: Ruta que capta el webhook lanzado desde Shopify

```
const express = require('express');
const app = express();

// Módulo algoritmos criptográficos seguros
const crypto = require('crypto');

// Clave secreta obtenida de las variables del entorno
const secretKey = process.env.shopify_secret;

// Ruta configurada en Shopify
app.post('/webhooks/orders/create', async (req, res) => {
  const hmac = req.get('X-Shopify-Hmac-Sha256');

  // Crea un hash usando el cuerpo y nuestra clave
  const hash = crypto
    .createHmac('sha256', secretKey)
    .update(req.body, 'utf8', 'hex')
    .digest('base64');

  // Compara el hmac con nuestro propio hash
  if (hash === hmac) {
    // En caso de corresponder obtenemos los datos y los almacenamos
    const order = JSON.parse(req.body.toString());
    // Llamada a la función que almacena las ordenes en Firestore
    orderUpdate(order);
    res.sendStatus(200);
  } else {
    // Esta solicitud no se originó en Shopify
    res.sendStatus(403);
  }
});
```

3.3.9. Websockets con Socket.IO

Si bien la base de datos Firebase Firestore proporciona una capa de conexión en tiempo real, es necesario introducir un nuevo método de comunicación permanente para notificar de eventos como lo son la presencia de usuarios activos y la visualización de las ordenes. Socket.IO permite la comunicación bidireccional entre el cliente y el servidor. Las comunicaciones bidireccionales se habilitan cuando un cliente tiene Socket.IO en el navegador, y un servidor también ha integrado el paquete. Si bien los datos se pueden enviar de varias formas, JSON es el más simple. Resume muchos tipos de transportes, incluidos AJAX y WebSockets, en una sola API. Permite a los desarrolladores enviar y recibir datos sin preocuparse por la compatibilidad entre navegadores [2].

En cualquier aplicación en tiempo real, mostrar múltiples usuarios en línea es muy importante, esta información debe actualizarse cuando un nuevo usuario se conecta o un usuario en línea se desconecta.

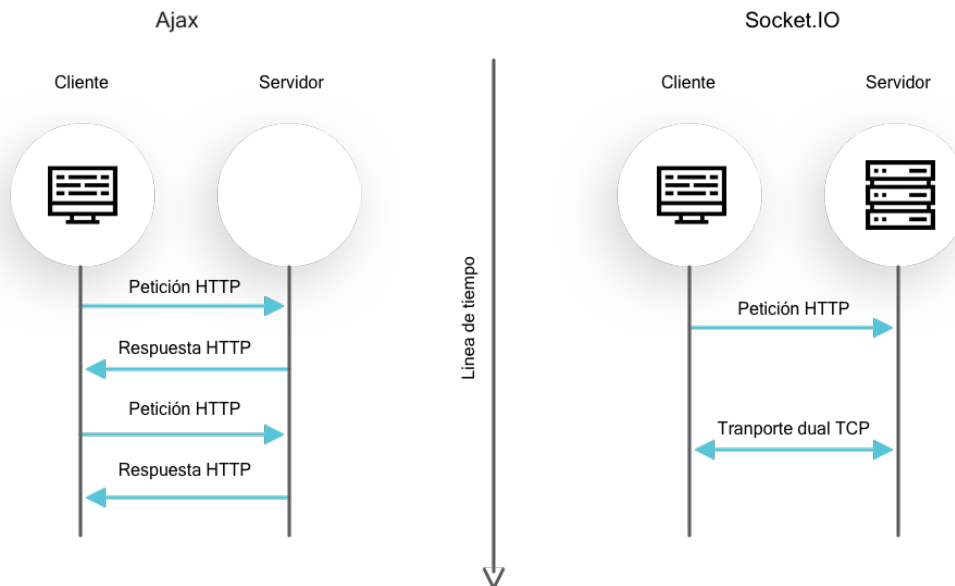


Figura 3.9: Comparación de transporte por sondeo (Ajax) y websockets (Socket.IO).

Código Ejemplo 3.9: Fragmento de configuración de Socket.IO del sistema

```
const express = require('express');
const app = express();
const http = require('http').Server(app);

// Inicializar sockets
const io = require('socket.io')(http);

// Variable para almacenar usuarios activos
const store = { users: {} };

// Ejecutar función cuando un cliente se conecte
io.on('connection', (socket) => {
  // Obtener token del cliente
  const { token } = socket.handshake.query;

  // Verificar y almacenar el id del socket
  if (store.users[token]) {
    // El cliente tiene varias ventans activas
    store.sockets[token][socket.id] = true;
  } else {
    // El cliente se conecta por primera vez
    store.sockets[token] = {};
    store.sockets[token][socket.id] = true;
  }

  // Emitir usuarios activos a todos los clientes
  io.emit('users_change', { data: store.users });

  // Ejecutar función cuando el cliente se desconecte
  socket.on('disconnect', () => {
    // Remover socket y emitir usuarios activos
    delete store.sockets[token][socket.id];
    io.emit('users_change', { data: store.users });
  });
});
```

3.4. Desarrollo de la aplicación web (Frontend)

3.4.1. Patrones de diseño

Los patrones de diseño facilitan la reutilización de diseños y arquitecturas exitosas. Los patrones de diseño ayudan a elegir alternativas de diseño que hacen que un sistema sea reutilizable y evitar alternativas que comprometan la reutilización. Pueden incluso mejorar la documentación y el mantenimiento de los sistemas existentes.

3.4.2. Redux

Redux es un administrador de estado predecible para aplicaciones JavaScript basado en el patrón de diseño Flux. A medida que una aplicación crece, se hace difícil mantenerla organizada y mantener el flujo de datos. Redux resuelve este problema administrando el estado de la aplicación con un único objeto global llamado Redux Store. Los principios fundamentales de Redux ayudan a mantener la coherencia en toda la aplicación, lo que facilita la depuración y las pruebas.

Redux/Flux

Redux adoptó un gran número de restricciones de la arquitectura Flux: las acciones encapsulan la información para que el Redux Reducer actualice el estado de manera determinista, el estado es un Redux Store singleton. El despachador de Flux único se reemplaza con múltiples Redux Reducers pequeños que recogen información de las acciones y la ‘reducen’ a un nuevo estado que luego se guarda en el Redux Store. Cuando se cambia el estado en el Store, la Vista según la suscripción recibe propiedades.

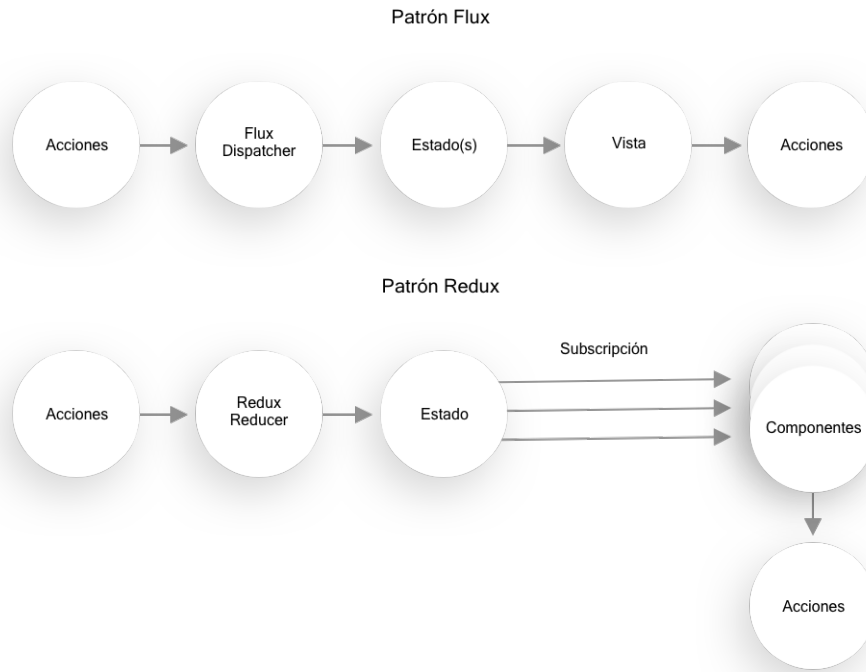


Figura 3.10: Comparación Flux/Redux.

Redux Store

Redux Store contiene un objeto del estado global de la aplicación. Esta actualiza el estado y notifica los componentes suscritos.

Código Ejemplo 3.10: Fragmento de código para inicializar el Redux Store

```
import { createStore } from 'redux';
import reducer from './reducer';

const store = createStore(reducer);
export default store;
```

Redux Reducer

Un Redux Reducer es solo una función pura de JavaScript. Recibe dos parámetros: el estado actual y la acción. Una función pura es aquella que devuelve exactamente la misma salida para la entrada dada. El estado es el objeto Redux Store completo, la acción es el objeto despachado con un tipo requerido y un payload opcional.

Código Ejemplo 3.11: Fragmento de código del reducer común de la app

```
const defaultState = { isAdmin: false };

export const reducer = (state = defaultState, action) => {
  switch (action.type) {
    case 'common/SET_ADMIN': {
      return { ...state, isAdmin: action.payload };
    }
    default: { return state; }
  }
};
```

Acciones Redux

La única forma de cambiar el estado es enviando una señal al Store. Esta señal es una acción. Entonces "despachar una acción" significa enviar una señal a Redux Store.

Código Ejemplo 3.12: Fragmento de una acción que actualiza el estado

```
export const setAdmin = payload => ({
  type: 'common/SET_ADMIN',
  payload,
});
```


3.5. Integración React/Redux

Para conectar el Store de Redux con React, es mediante un componente llamado Provider. El único propósito de Provider es agregar el Store al contexto del componente de la Aplicación, para que todos los componentes secundarios puedan acceder a ella. Provider envuelve a la aplicación React y hace que sea consciente de el Store.

Código Ejemplo 3.13: Fragmento de código de la aplicación React principal

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';

// La interfaz común para todos los componentes de enrutamiento
import { Router } from 'react-router-dom';
// Módulo para generar urls basadas en hash
import { createHashHistory } from 'history';
// Store de la aplicación
import store from '../core/redux/store';
// Componentes de enrutamiento e interfaz general
import Layout from '../core/layout';
import Routes from '../core/routes';
// Crear historia
const history = createHashHistory();
// Rederizar app
ReactDOM.render(
  <Provider store={store}>
    <Router history={history}>
      <Layout>
        <Routes />
      </Layout>
    </Router>
  </Provider>,
  document.getElementById('Root'), // Elemento del DOM
);
```

Bibliografía

- [1] Firebase. *Transacciones y escrituras en lotes*. 2019. URL: <https://firebase.google.com/docs/firestore/manage-data/transactions>.
- [2] Fionn Kelleher. *Understanding Socket.IO*. 2014. URL: <https://nodesource.com/blog/understanding-socketio/>.
- [3] Azat Mardan. *Express.js Guide: The Comprehensive Book on Express.js*. 2014. URL: <https://pepa.holla.cz/wp-content/uploads/2016/08/Express.js-Guide.pdf>.
- [4] *Node.js web application framework*. URL: <http://expressjs.com/>.
- [5] Shopify. *Storefront API*. 2019. URL: <https://help.shopify.com/en/api/storefront-api>.
- [6] Shopify. *Webhooks — Shopify Help Center*. 2019. URL: <https://help.shopify.com/en/api/reference/events/webhook>.