INSTITUTO METROPOLITANO DE PLANEACIÓN

DISEÑO Y DESARROLLO DE CAMINAPP

POR

JOSÉ LUIS MURILLO RÍOS

TIJUANA, B.C. AGOSTO 2019

DISEÑO Y DESARROLLO DE CAMINAPP

POR

JOSÉ LUIS MURILLO RÍOS

AGOSTO 2019

ABSTRACT

This document is aimed at people with intermediate knowledge in programming and a basic reasoning of the operation of the web, here it is described the development of a point of sale system, formed by a server based on Node.js and a web application created with ReactJS , using the most modern tools and procedures for web development.

RESUMEN

Este documento está dirigido a personas con conocimientos intermedios en programación y un razonamiento básico del funcionamiento de la web, aquí se describe el desarrollo de un sistema de punto de venta, formado por un servidor basado en Node.js y una aplicación web creada con ReactJS, utilizando las herramientas y los procedimientos mas modernos para el desarrollo web.

Índice general

Índice de tablas

Índice de figuras

Código Ejemplo

Capítulo 1

Introducción

1.1. Antecedentes y definición del problema

La capacidad de caminar se ha reconocido cada vez más como un factor importante para el desarrollo urbano sostenible que, sin embargo, rara vez se ha investigado en ciudades de rápida urbanización. La caminabilidad captura la proximidad entre los usos del suelo funcionalmente complementarios o la conectividad entre destinos desde la perspectiva de accesibilidad, que a menudo se asocia con factores como el ancho de la calle, la conexión de la calle, los cruces peatonales, el número de carriles, velocidades seguras, etc. Además de la perspectiva de accesibilidad, la capacidad de caminar también se puede definir como un ambiente agradable para caminar, que está influenciado por una serie de factores, como la limpieza de las calles, los cruces seguros, la sensación de seguridad, la apariencia de los árboles de la calle, la iluminación nocturna, etc.

Por tales motivos se debe contar con un sistema fácil de usar para usuarios de distintas áreas de la ciudad 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 asi como un servidor que administre los permisos adecuados para su manipulación.

1.2. Propuesta de solución

Implementar un sistema diseñado principalmente para ser utilizado en pantallas táctiles que muestre y capture propuestas de mejora o reportes de caminabilidad, dando un informe detallado de ellos.

1.3. Objetivos generales

Automatizar el proceso de reportes de caminabilidad mediante una aplicación web progresiva, que sea compatible con los dispositivos existentes de la empresa y que funcione adecuadamente en navegadores de internet modernos y en dispositivos móviles.

1.4. Objetivos específicos

1. Desarrollar un servidor que administre la autenticación de los usuarios 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

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* 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 *frontend*, 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 [13].

2.1.1. Componentes MEAN

- MongoDB (base de datos)
  Una opción increíblemente popular en el mundo de la gestión de bases de datos NoSQL.
- Express.js (servidor)
  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 *frontend*.
- Angular.js / ReactJS (cliente)
  Es un potente *framework frontend*, utiliza un patrón de diseño Modelo-Vista-Controlador. ReactJS es una opción alternativa para el desarrollo *frontend*, aunque React es simplemente una biblioteca, no un *framework* MVC completo.
- Node.js (entorno del servidor)
  Permite al programador escribir el *backend* de la aplicación en Javascript y ejecutarlo en la mayoría de los sistemas operativos modernos.

Derivados:

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

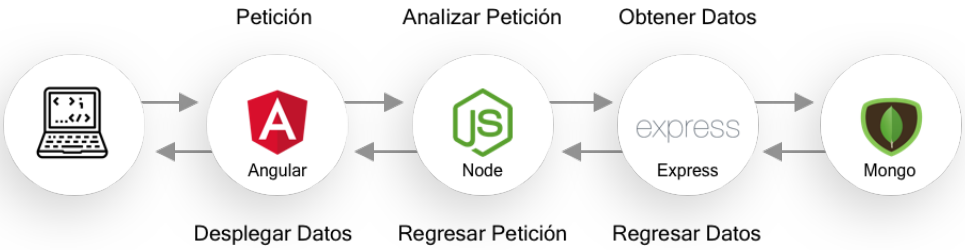


Figura 2.1: Flujo de información en MEAN Stack. (Fuente: Elaboración propia)

2.1.2. 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.

2.1.3. 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.

2.1.4. Base de datos NoSQL

A pesar de la gran cantidad de bases de datos SQL, también hay una tendencia hacia NoSQL. Las bases de datos NoSQL permiten almacenar datos no estructurados y heterogéneos. La *escalabilidad* mejorada ha ayudado a aumentar su popularidad en mayor medida en el mercado actual. El escalado horizontal significa que la organización no tiene que preocuparse por la infraestructura subyacente. 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.

	SQL	NoSQL
Tipo	Relacional	Distribuido
Datos	Datos estructurados almacenados en tablas	Datos no estructurados almacenados en archivos JSON
Esquema	Estático o predefinido	Dinámico
Escalable	Vertical	Horizontal
Consultas	Adecuado para consultas complejas	Lenguaje de consulta no estructurado
Flexible	Esquema rígido ligado a la relación	Esquema adecuado para almacenamiento jerárquico
Elasticidad	Requiere tiempo de inactividad en la mayoría de los casos	Automática, no requiere interrupción

Tabla 2.1: Comparativa SQL y NoSQL

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 las bases de datos relacionales (RDB), 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 *RDBMS*.

2.1.5. 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).

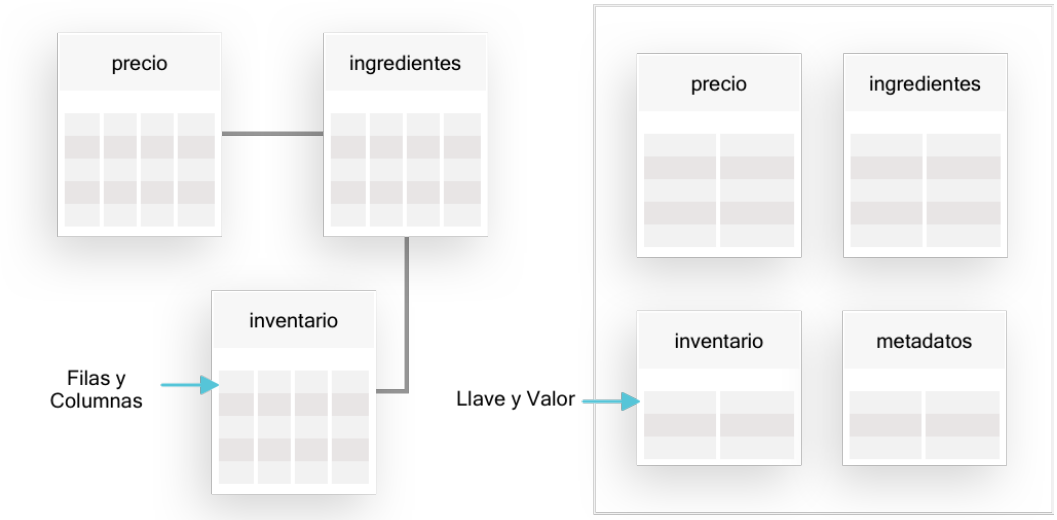


Figura 2.2: Diagrama que representa las diferencias clave entre la base de datos SQL y las bases de datos NoSQL. (Fuente: Elaboración propia)

2.1.6. Angular/React

Angular o React, proporcionan la interfaz de usuario reactiva de una 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).

	Angular	React
Desarrollador	Google	Facebook
Definición	*Framework*	Librería
Modelo de plantilla	HTML + Typescript	JSX + Javascript
Flujo	2 vías	Unidireccional
DOM	Regular	Virtual
Lógica/Estado de la aplicación	Services	Flux/Redux

Tabla 2.2: Características de Angular.js y ReactJS

Angular es un *framework* con muchas herramientas integradas, para hacer solicitudes HTTP, enrutamiento y navegación, animaciones y otros. Se basa en módulos que son componentes y servicios.

React es una biblioteca de Javascript, que se puede usar para crear nuevas aplicaciones o para integrarla con una aplicación existente. React se basa en componentes pequeños y reutilizables, que administran su propio estado y luego los componen para crear interfaces de usuario complejas. Incluso si React no es tan complejo como Angular, con muchas cosas integradas, hay muchas bibliotecas que se pueden agregar para tener enrutadores (react-router) y solicitudes HTTP (*axios*), manejo de estado (react-redux) entre otras más. Esto lo hace portátil y fácil de incorporar en cualquier entorno.

2.2. FERN

Firebase es una plataforma propiedad de Google que tiene como objetivo proporcionar un enfoque completo 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 NoSQL, alojamiento, almacenamiento de archivos, procesamiento del lado del servidor para cosas que deben protegerse de la interfaz y un sistema de autenticación, que junto a módulos de desarrollo *frontend* como ReactJS, proporcionan todo lo necesario para crear aplicaciones web pequeñas y medianas.

2.2.1. Componentes FERN

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

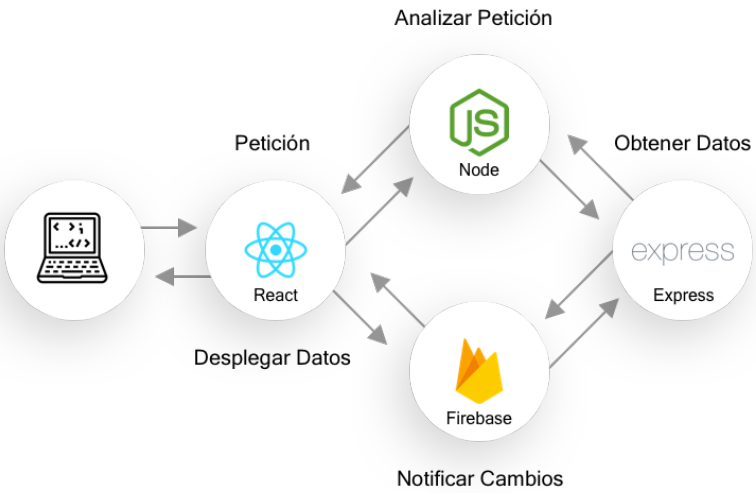


Figura 2.3: Flujo de información en FERN Stack. (Fuente: Elaboración propia)

2.2.2. Cloud Firestore

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 aplicaciónes 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.

Firestore es una base de datos orientada a documentos, toda la información se guarda en colecciones como en un 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.

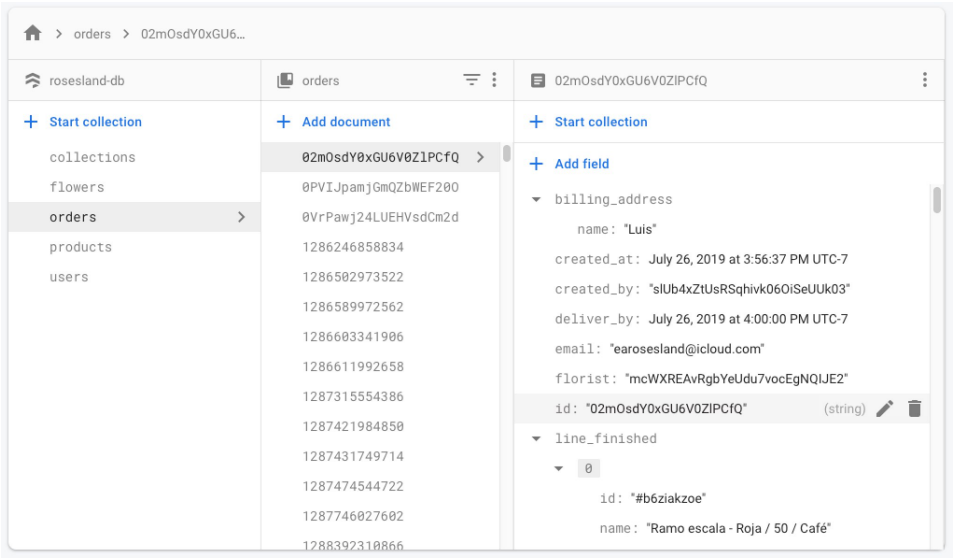


Figura 2.4: Colección de Firestore con sus documentos internos. (Fuente: Elaboración propia)

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.

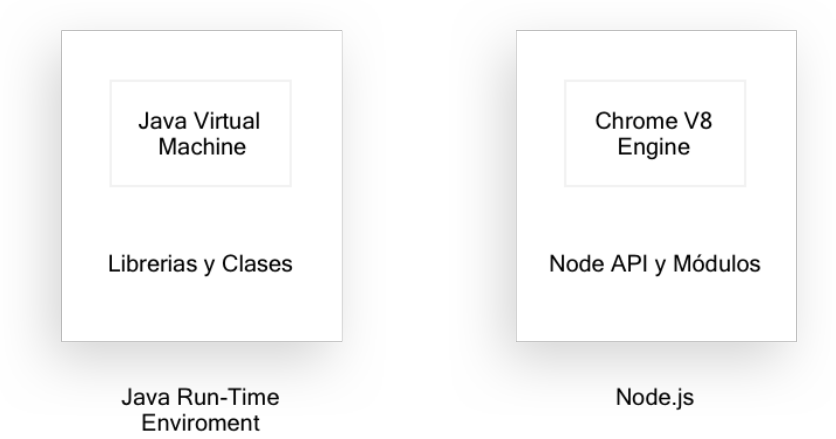


Figura 2.5: Analogía Node.js con Java. (Fuente: Elaboración propia)

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.

Navegador	Motor de *renderizado* / diseño	Motor de secuencias de comandos
Chrome	Blink (C++)	V8 (C++)
Mozilla Firefox	Gecko (C++)	SpiderMonkey (C/C++)
IE Edge	EdgeHTML (C++)	Chakra JavaScript engine (C++)
Opera	Blink (C++)	V8 (C++)
Internet Explorer	Trident (C++)	Chakra JScript engine (C++)

Tabla 2.3: Motores de renderizado y secuencias de comandos

2.2.4. El ecosistema NPM

NPM (Node Package Manager) es el administrador de paquetes predeterminado para Node.js. Paquete es un término utilizado por npm para denotar herramientas que los desarrolladores pueden usar para sus proyectos [5]. 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 especi-

ficación 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. ReactJS

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 términos generales, al crear aplicaciones con ReactJS, 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 comúnmente involucran datos complejos y manejo de estado. ReactJS 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 estos elementos. Son similares, en conceptos, a cosas como *widgets* y módulos. React se define a sí mismo como una biblioteca para construir interfaces de usuario. Como tal, cuando se piensa en una interfaz de usuario, se debe pensar en ella en términos de los componentes más pequeños posibles que se puedan definir. La razón por la que existe este paradigma es para disminuir el acoplamiento (cuánto dependen los unos de los otros) y aumentar la cohesión (qué tan bien funcionan juntas las diferentes cosas).

Cada uno de estos fragmentos es un bloque de código independiente y reutilizable, que divide la interfaz de usuario en partes más pequeñas. Incluyen código que define cómo se crean los elementos en el *DOM* y cómo los usuarios pueden interactuar con ellos. Los componentes se pueden definir únicamente en JavaScript o se pueden definir en un superconjunto (o variación extendida) de JavaScript llamado JSX, que se explica en temas posteriores.

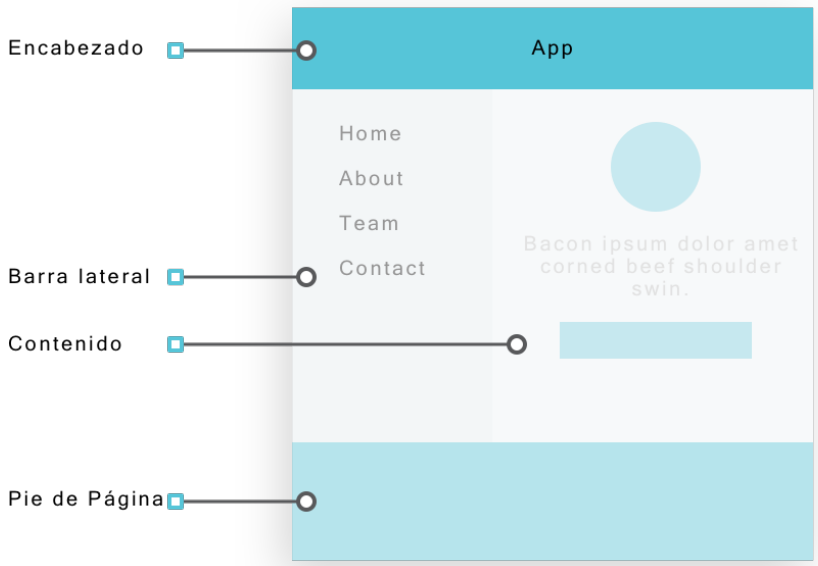


Figura 2.6: Componentes principales de una página web. (Fuente: Elaboración propia)

En primer lugar, hay un elemento principal llamado componente APP. Este componente de la aplicación contiene cuatro fragmentos 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. En pocas palabras, un componente es una clase o función de JavaScript que opcionalmente acepta entradas, es decir, propiedades (props) y devuelve un elemento que describe cómo debería lucir una sección de la interfaz de usuario.

Código Ejemplo 2.2: Ejemplo de página con componentes ReactJS

```
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

En el desarrollo web de software, *DOM* significa Modelo de Objeto de Documento (Document Object Model) y representa la estructura de los elementos en una página web. El HTML DOM se construye como un árbol de objetos.

Algunas bibliotecas de JavaScript, como jQuery, manipulan los elementos DOM directamente, cambiando sus atributos, agregando o eliminando componentes. Sin embargo, en lugar de cambiar el DOM directamente, React usa un DOM virtual, que es una replicación virtual del árbol DOM actual. En consecuencia, React puede manipular el DOM virtual innumerables veces y comparar su estado con el DOM real. De esta manera, React sabe exactamente qué elemento cambió y luego actualiza solo ese elemento específico en la pantalla.

Detrás de escena, React hace un gran trabajo para editar y volver a *renderizar* eficientemente el DOM cuando algo en la interfaz necesita cambiar. Este enfoque le brinda al desarrollador una gran flexibilidad y sorprendentes ganancias de rendimiento porque ReactJS calcula con anticipación qué cambios se deben realizar en el DOM y actualiza los árboles DOM en consecuencia. De esta manera, ReactJS evita las costosas operaciones DOM y realiza actualizaciones de manera eficiente [12].

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 Eyplorer 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.

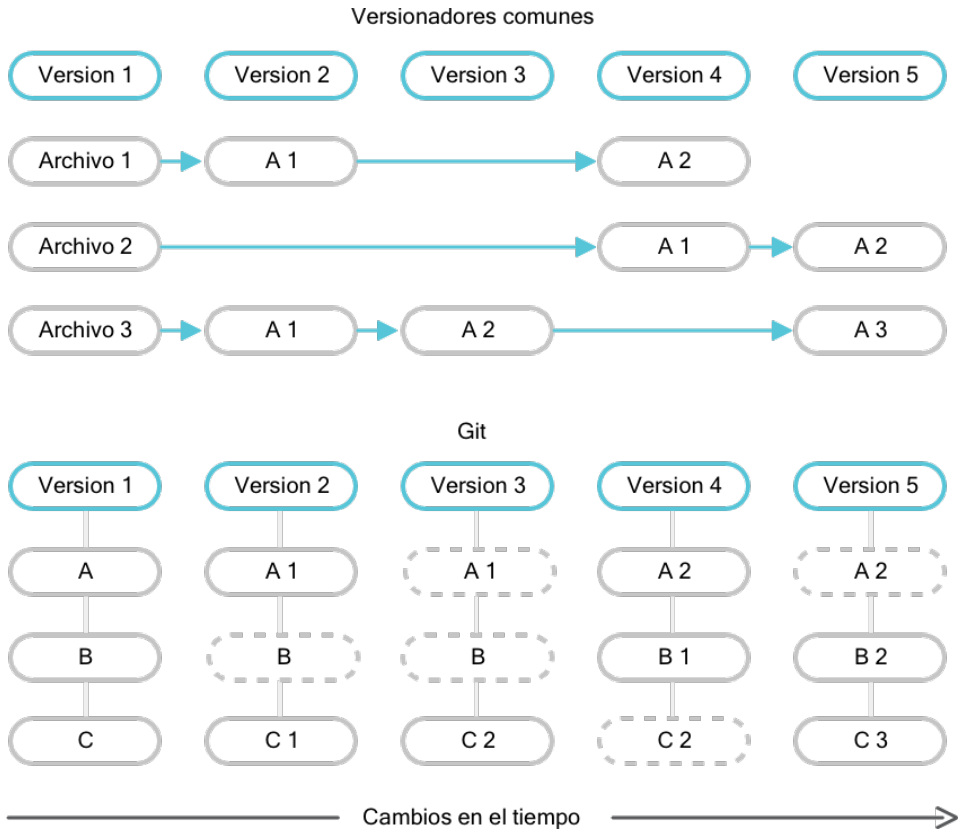


Figura 2.7: Diferencias entre controladores de versiones. (Fuente: Elaboración propia)

Capítulo 3

Análisis y diseño del sistema

A continuación se describen los procedimientos necesarios para elaborar un sistema que permita elaborar y mostrar sugerencias de caminabilidad en el área de tijuana, así como incorporar servicios de geolocalización..

3.1. Requerimientos del proyecto

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.

3.1.1. 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 lo 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 *frontend* 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 modulo 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 *frontend* probablemente se minimizará y se enviará al navegador y es público en su naturaleza básica. Y el lado del servidor o el *backend* 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 *backend*.
- 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

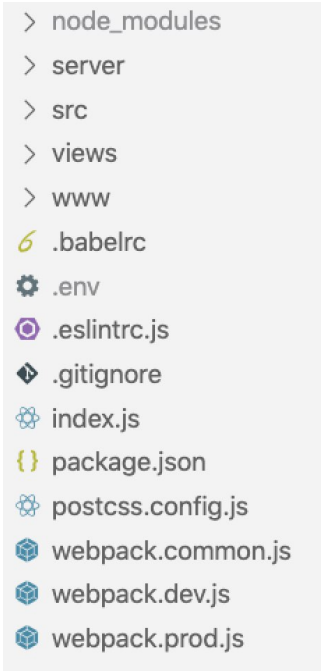


Figura 3.1: Vista general de la raíz del proyecto. (Fuente: Elaboración propia)

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=""
```

3.2. 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.

3.2.1. 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.

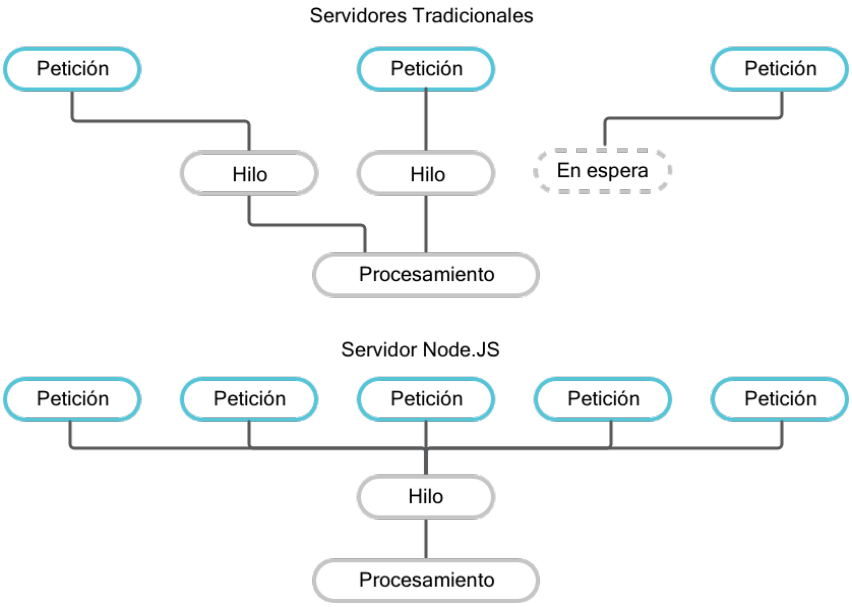


Figura 3.2: Comparación de node y servidores tradicionales. (Fuente: Advance Idea Infotech 2018)

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 [1].

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!');
 // Indiqua 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!'.

3.2.2. Configuración Express

Express.js es un 'marco de aplicación web NodeJS minimalista y flexible' [10]. 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 [7].

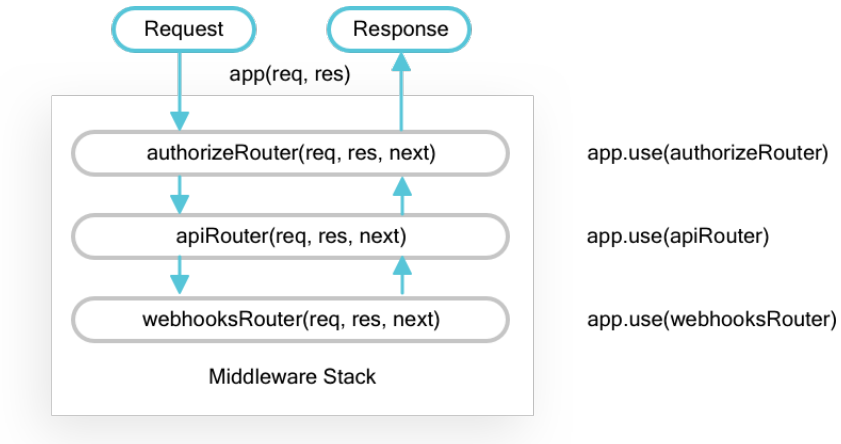


Figura 3.3: Cada función de middleware en la pila se ejecuta antes que las que están debajo de ella. (Fuente: Elaboración propia)

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 analisís 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) => {...});
```

3.2.3. 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.

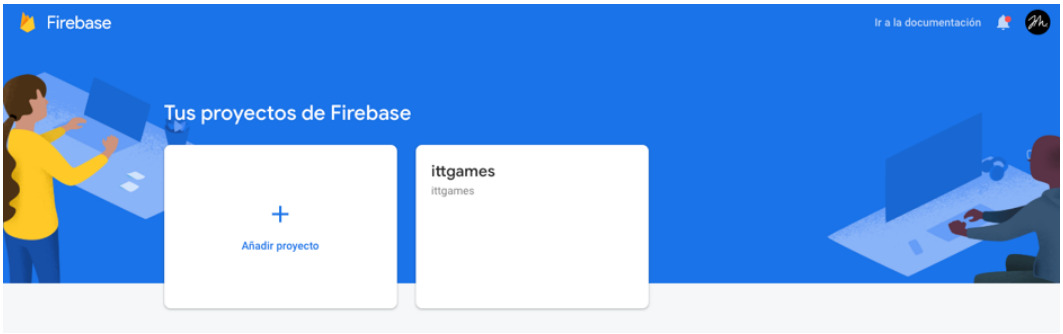
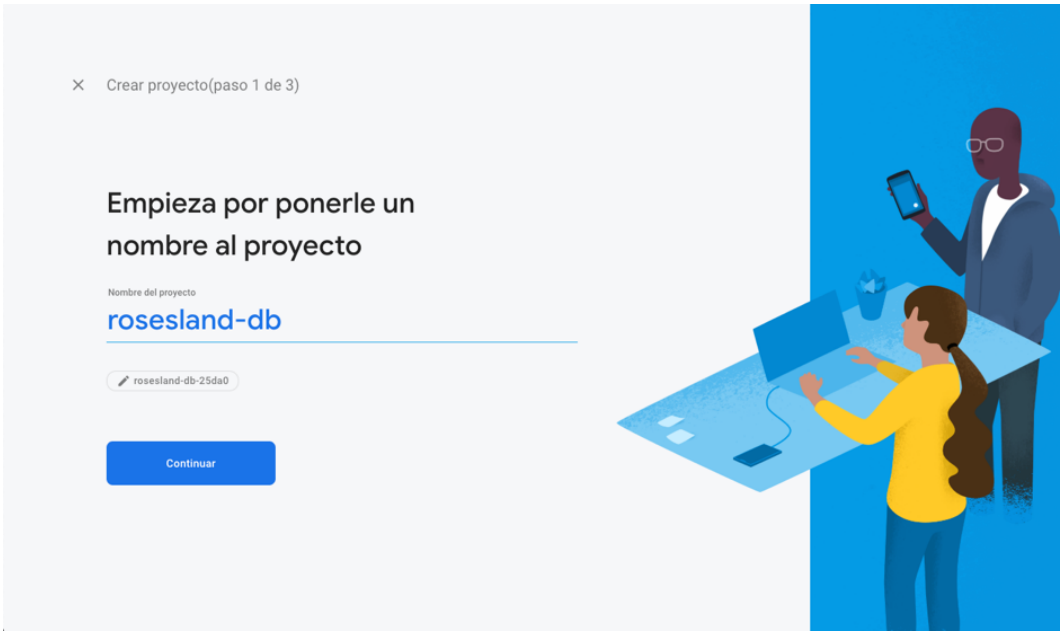
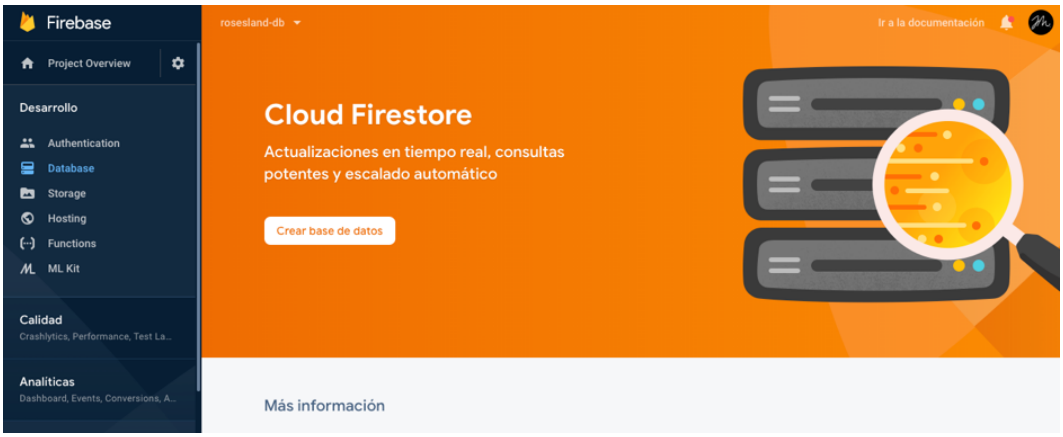


Figura 3.4: Consola de Google Firebase. (Fuente: Google Console)

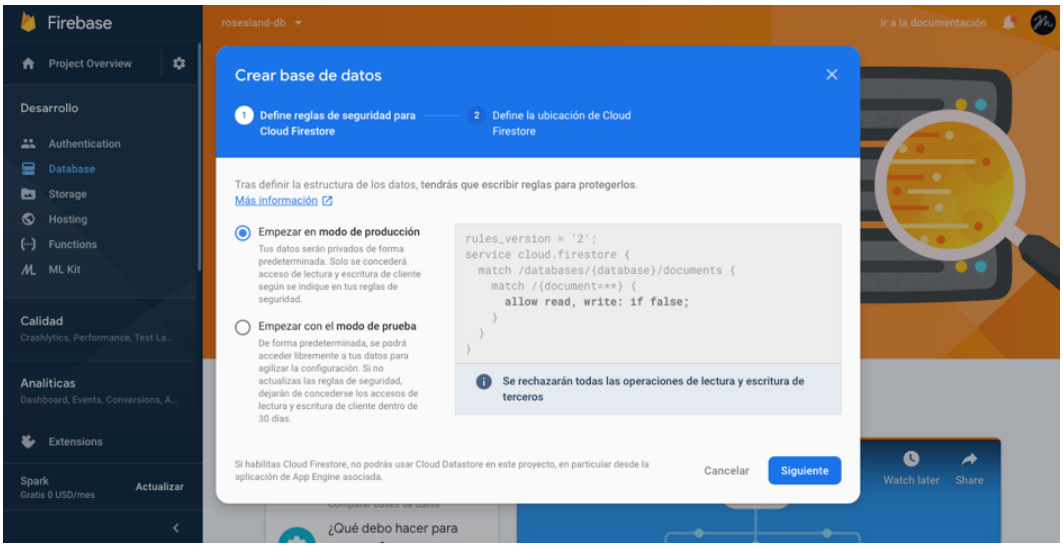
1. Crear un nuevo proyecto en la consola de Firebase.



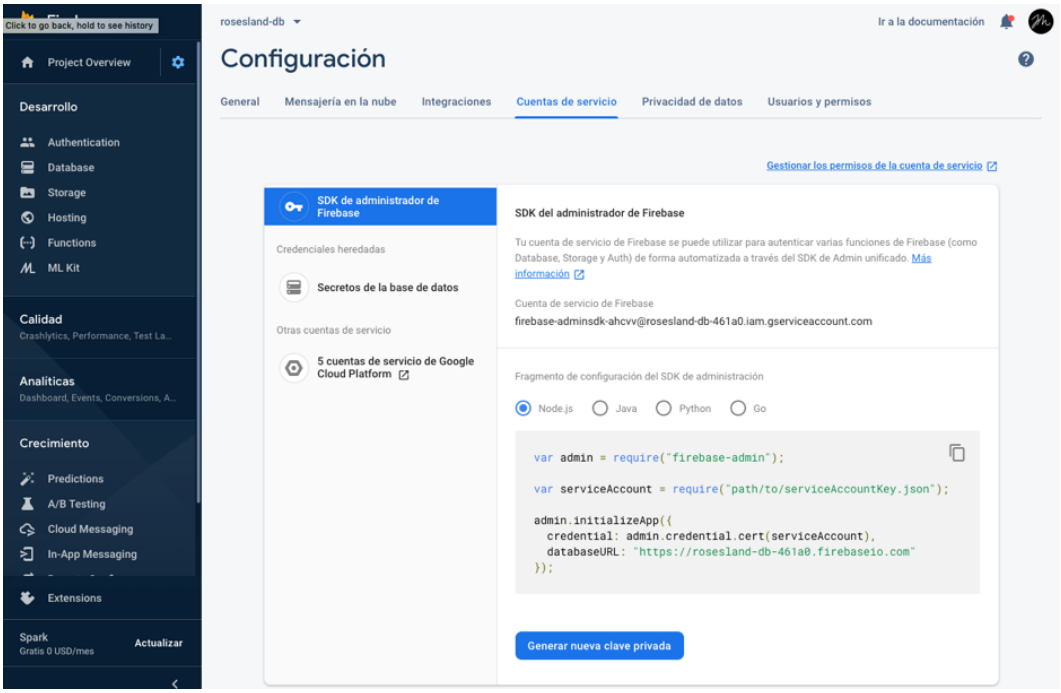
2. En el apartado 'Database', seleccionar 'crear base de datos' .



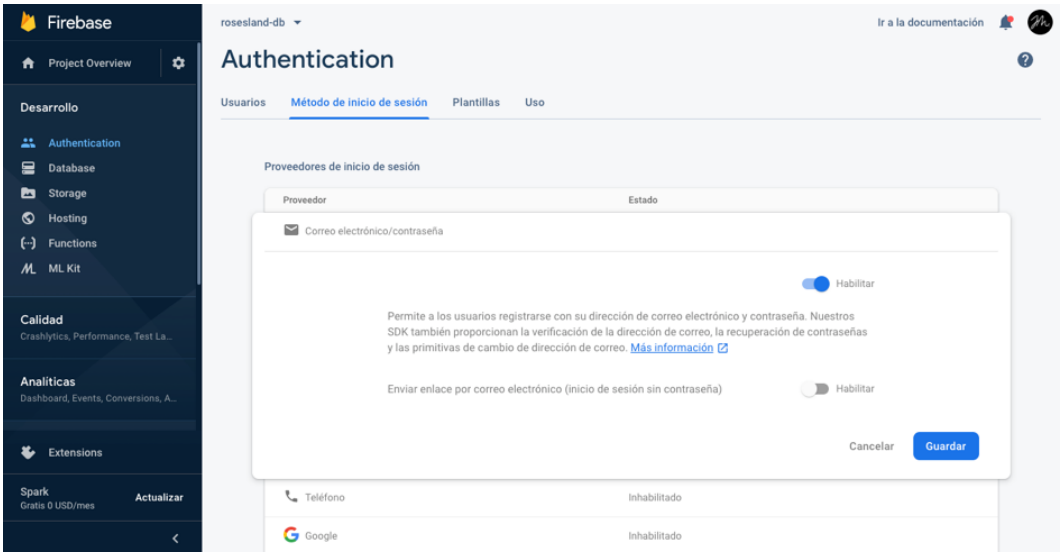
3. En la ventana emergente, seleccionar reglas de seguridad y definir ubicación del servidor.



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 [3]:

Código Ejemplo 3.5: Fragmento para guardar datos en Firestore

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

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

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

// Función que guarda productos en al 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 funcion 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);
    });
};
```

3.2.4. 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 los ejecute.

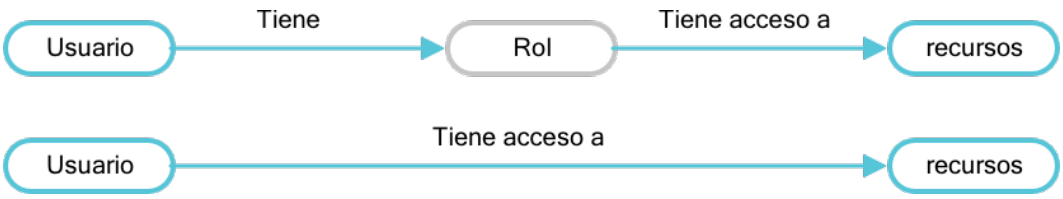


Figura 3.5: Autenticación basada en roles. (Fuente: Elaboración propia)

Firebase Custom Claims

Los roles de usuario son necesarios para identificar a los usuarios como administradores, gerentes o simplemente como clientes. Firebase 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 peticón.
        next();
      })
      .catch((error) => {
        res.status(400).send({ data: 'error', message: error });
      });
  }
});
```

3.2.5. 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 [6].

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.

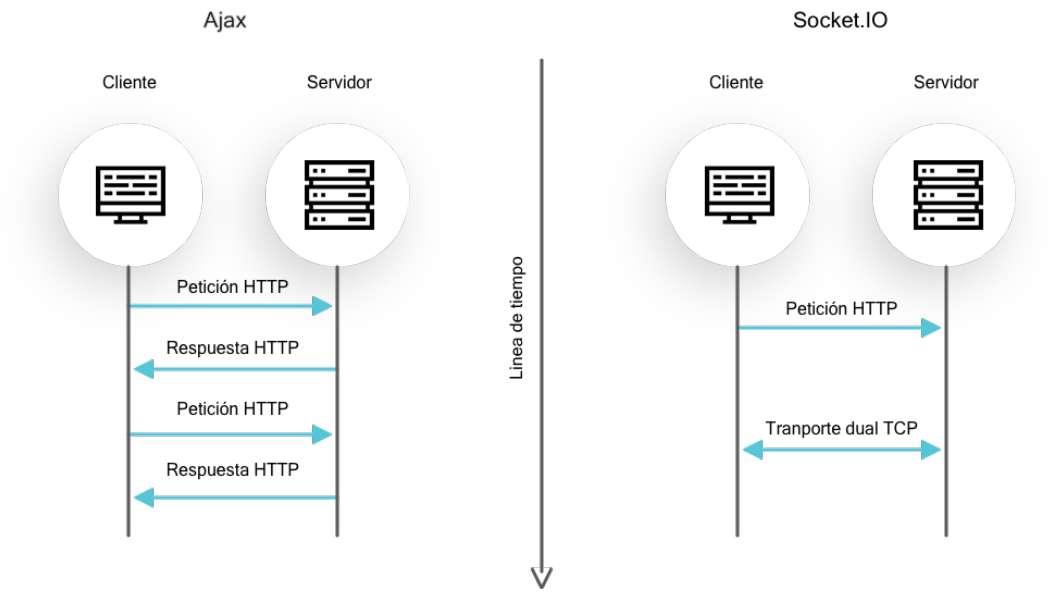


Figura 3.6: Comparación de transporte por sondeo (Ajax) y Websockets (Socket.IO).

Código Ejemplo 3.7: 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 });
  });
});
```

3.3. Desarrollo de la aplicación web (Frontend)

El desarrollo de la aplicación web presenta grandes desafíos, debido a que en la actualidad existen un gran número de dispositivos móviles y navegadores web con características muy diferentes. Una buena aplicación web es más que algo para mirar, es funcional, interactiva e impecable. A medida que las tecnologías se vuelven inteligentes, debemos ser lo suficientemente inteligentes como para utilizarlas. Con la rápida evolución de las tecnologías web, la complejidad de las aplicaciones web también ha crecido, especialmente al hacer una aplicación web que funcione bien con todas las versiones de todos los navegadores [11].

El desarrollo web es un campo en constante cambio: la forma en que construimos sitios web hoy en día es completamente diferente de cómo solíamos hacerlo hace un par de años. Con la gran cantidad de herramientas disponibles y las nuevas que aparecen todos los días, la mayoría de las veces los desarrolladores se sienten confundidos sobre qué camino tomar.

3.3.1. Configuración del entorno de desarrollo

Los módulos en JavaScript, como en cualquier otro lenguaje de programación, ayudan a descomponer código en partes separadas más pequeñas. Este patrón ayuda a gestionar la creciente complejidad al mantener las preocupaciones separadas en sus partes independientes. En pocas palabras, los módulos ayudan a organizar el código.

Antes de ES6, esto se podía hacer usando diferentes archivos de *script* y luego cargando cada uno de ellos por separado con una etiqueta `<script>` en nuestro HTML. Esto tenía muchas desventajas, como mantener el orden correcto de los *script* para evitar romper accidentalmente cualquier código dependiente. Pero afortunadamente, ES6 trajo soporte para módulos con las

palabras clave `import` y `export`, pero aún no son totalmente compatibles en todos los entornos (navegador y Node.js).

Webpack le permite escribir su código en módulos y unificarlos en uno o más paquetes. Webpack permite usar módulos y todas sus bondades sin preocuparse por el soporte. Además de JavaScript, también puede incluir otros tipos de archivos, incluidos (pero no limitados a) CSS, fuentes, imágenes, HTML, etc. y luego transformarlos en un formato aceptable. Webpack es extremadamente potente y se puede ampliar para hacer cosas impresionantes usando el concepto de loaders y *plugins*.

Conceptos clave

Webpack necesita una configuración básica para que funcione debidamente:

- Entrada: Webpack utiliza el grafo de dependencias para decidir qué módulos deben agruparse. Esto significa que Webpack comienza desde un solo módulo y procesa todas sus dependencias directas e indirectas para formar el grafo de dependencia completo y luego agrupar todos los módulos necesarios. El punto de entrada determina desde dónde debe comenzar el paquete web para construir su gráfico de dependencia interna.
  Entrada del proyecto: `./src/app.js`
- Salida: La salida determina dónde se supone que el paquete web debe emitir los paquetes que crea y cómo los nombra. Este directorio también contendrá todos los archivos estáticos de la aplicación web que serán visibles al publico mediante el servidor Express.js.
  Salida del proyecto: `./www/bundle.js`
- Loaders Los loaders en el paquete web son los que le permiten manejar archivos que no son JavaScript (el empaquetador web solo comprende JavaScript). Los loaders leen varios tipos de archivos y los transforman en módulos válidos que webpack puede entender.

- Plugins Los *plugins* son la característica más poderosa de webpack, se utilizan para una amplia gama de tareas que los loaders no pueden realizar. Se utilizan para la optimización de paquetes, *minificación*, emisión de estadísticas, etc.

Babel Loader

Babel ofrece el último soporte de sintaxis ECMAScript (ES5, ES6). Las librerías mas recientes insisten en que se utilicen las últimas ofertas de JavaScript para obtener un código más limpio y legible. Pero desafortunadamente nuestros navegadores no entienden la mayor parte de la sintaxis aquí es donde entra en juego Babel. Es responsable de convertir el código ES5 y ES6 en código comprensible por el navegador, básicamente compatibilidad con versiones anteriores. Las dependencias que necesita el proyecto son las siguientes:

- babel-core: El motor principal de Babel para que sus dependientes trabajen.
- babel-preset-env: esta es la parte de soporte ES5, ES6
- babel-preset-react: Babel se puede usar en cualquier *framework* que necesite el soporte de sintaxis JS más reciente, en este caso es 'React'.
- babel-loader: Un puente de comunicación entre Webpack y Babel

Código Ejemplo 3.8: Fragmeno de configuración Babel.

```
{
  "presets": ["env", "react"],
}
```

Código Ejemplo 3.9: Fragmeno de configuración Webpack

```
const path = require('path');
const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  entry: path.resolve(__dirname, 'src', 'app.js'), // Entrada principal

  output: {
    path: path.resolve(__dirname, 'www'), // Directorio de salida
    filename: 'bundle.js',  // Nombre de archivo de salida
  },

  module: {
    rules: [
      {
        test: /.js$/,
        loader: 'babel-loader', // Comunicación entre Webpack y Babel
        ...
      },
      {
        test: /.(gif|png|jpg|jpeg)$/,
        loader: 'file-loader',  // Loader de archivos
        ...
      },
      {
        test: /.scss$/, // Loaders de hojas de estilo
        use: [
          "style-loader",
          "css-loader",
          "sass-loader"
        ]
      },
    ],
  },
  // Plugin que extrae texto a un archivo separado
  plugins: [new ExtractTextPlugin(path.join('bundle.css'))],
};
```

Webpack Dev Server

La herramienta Webpack Dev Server ejecuta y sirve una compilación de nuestro proyecto, pero no lo escribe en el disco, lo hace en la memoria. En modo de desarrollo, este modulo hace que la aplicación se abra un navegador, y vuelva a cargar el navegador cuando detecte cambios en los archivos de los que depende la aplicación web.

Código Ejemplo 3.10: Configuración Webpack Dev Server.

```
const path = require('path');
// Módulo para unir la confuguración común de webpack
const merge = require('webpack-merge');
// Confuguración común
const common = require('./webpack.common.js');

module.exports = merge(common, {
  mode: 'development', // Modo de desarrollo
  devtool: 'inline-source-map',

  devServer: {
    // Directorio de entrada
    contentBase: path.join(__dirname, 'www'),
    compress: true,
    port: 8002, // Puerto de desarrollo

    historyApiFallback: true,
    hot: true,
    host: '0.0.0.0',
    // Proxy para conexión con el backend
    proxy: [{
      context: ['/socket.io', '/api'],
      target: 'http://localhost:3000',
    }],
  },
});
```

Con la ayuda de Webpack se tiene la capacidad de importar estilos y componentes de React y hacer que algo no solo sea funcional, sino también estético. Los procesos de construcción pueden ser desalentadores, pero si se tiene una base sólida y se sabe cómo expandirla, pueden obtenerse buenos resultados.

3.3.2. Patrones de diseño

Las aplicaciones web modernas tienen una estructura de programa compleja, debido a las funcionalidades que proporcionan en sus interfaces de usuario. Escribir manualmente un código de programa puede dar como resultado una calidad y contenido desiguales en partes individuales de la aplicación. Mantener tales aplicaciones desarrolladas es más difícil. Debido a esto, las aplicaciones web a menudo se desarrollan utilizando diferentes *frameworks* y patrones de diseño.

Estos esquemas facilitan la reutilización de diseños y arquitecturas exitosas; también 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. Estos estándares de resolución de problemas pueden ser increíblemente útiles

si se usan en las situaciones correctas y por las razones correctas. Cuando se usan estratégicamente, pueden hacer que un programador sea significativamente más eficiente al permitirle el usar métodos refinados por otros y evitar 'reinventar la rueda'. También proporcionan un lenguaje común útil para conceptualizar problemas y soluciones repetidos cuando se discute con otros o se maneja el código en equipos más grandes.

3.3.3. 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 se puede conectar con cualquier biblioteca de JavaScript. Sin embargo, funciona muy bien con ReactJS debido a su naturaleza funcional.

Redux ayuda a separar el estado de la aplicación, crea un almacén global que reside en el nivel superior de una aplicación y alimenta con el estado a todos los componentes internos. A diferencia de Flux, Redux no tiene múltiples objetos de almacenamiento. El estado completo de la aplicación está dentro de un objeto, y potencialmente podría intercambiar la capa de vista con otra biblioteca con el almacenamiento intacto.

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 llamadas props.

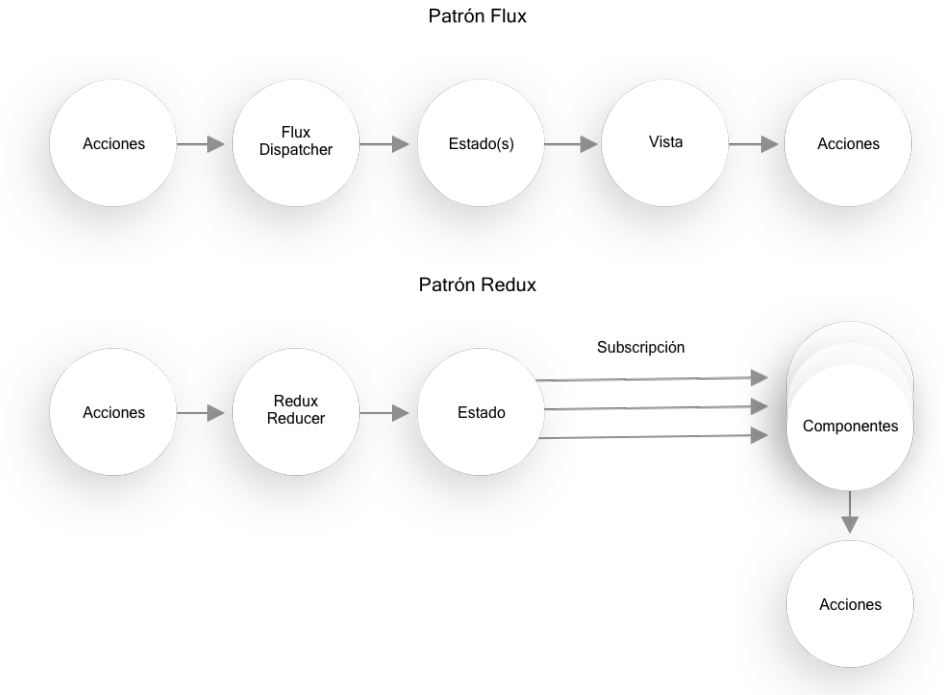


Figura 3.7: Comparación Flux/Redux. (Fuente: Elaboración propia)

Para trabajar con Redux se necesitan tres cosas:

- Actions (acciones): estos son objetos que deben tener dos propiedades, una que describe el tipo de acción y otra que describe lo que se debe cambiar en el estado de la aplicación.
- Reducers (reductores): son funciones que implementan el comportamiento de las acciones. Cambian el estado de la aplicación, en función de la descripción de la acción y la descripción del cambio de estado.
- Store (almacén): reúne las acciones y los reducers, manteniendo y cambiando el estado de toda la aplicación; solo hay una store.

Redux Store

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

Código Ejemplo 3.11: 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 3.12: 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 3.13: Fragmento de una acción que actualiza el estado

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

Este es un modelo conveniente y directo para estructurar datos en una aplicación y presentarlos en el cliente. La aplicación tiene un estado raíz. Un cambio de estado desencadena actualizaciones de vista. Solo las funciones especiales pueden modificar el estado. Una interacción del usuario activa estas funciones especiales de cambio de estado. Solo se produce un cambio a la vez. Esto significa que el estado central no puede desencadenar ninguna otra acción. Solo una entrada del usuario puede desencadenar otra acción [9].

3.3.4. Integración React/Redux

Redux practica la teoría de flujo de datos unidireccional y se convirtió en un patrón de facto como tecnología de gestión de estado para aplicaciones ReactJS. React utiliza *props* (abreviatura de propiedades) en un componente que permite el uso de variables no estáticas. Con la ayuda de *props*, podemos pasar estas variables a varios otros componentes (secundarios) desde el componente principal.

La conexión del Store de Redux con los componentes de React, es mediante un componente llamado `Provider` del módulo `react-redux`. En React para

compartir datos entre componentes, un estado tiene que vivir en el componente principal. Este componente principal proporciona un método para actualizar este estado y se pasa como *props* a estos componentes. 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 mediante la función `connect` de `react-redux`. `Provider` envuelve a la aplicación React y hace que sea consciente de el Store.

Código Ejemplo 3.14: 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
);
```

La función `connect` ayuda a conectar un componente con el estado de la aplicación, se debe definir una función especial llamada `mapStateToProps` para describir que partes del estado actual de Redux se desean pasar al componente que está envolviendo y una función de nombre `mapDispatchToProps` conecta las acciones de Redux con React *props*. De esta manera, un componente React conectado podrá enviar mensajes a el Store.

Código Ejemplo 3.15: Ejemplo de conexión de un componente React con el estado Redux

```
import React from 'react';
// Módulo de conexión
import { connect } from 'react-redux';
// Acciones Redux
import { loginUser } from '../../../redux/actions/common/async';

const Home = ({ user, login }) => (
  <div>
    <button onClick={login}>{user ? user.name : 'Login'}</button>
  </div>
);

// Conecta las partes del estado que se van a utilizar
const mapStateToProps = (state) => ({
  user: state.common.user,
});

// Inyecta los metodos necesarios para actualizar el estado
const mapDispatchToProps = (dispatch) => ({
  login: () => {
    dispatch(loginUser())
  },
});

// Vincula el componente con el estado
export default connect(mapStateToProps, mapDispatchToProps)(Home);
```

3.3.5. Diseño Atómico (Atomic Design)

El diseño atómico es una metodología compuesta por cinco etapas distintas que trabajan juntas para crear sistemas de diseño de interfaz de una manera más deliberada y jerárquica. Las cinco etapas del diseño atómico son:

1. Átomos
2. Moléculas
3. Organismos
4. Plantillas
5. Páginas

El diseño atómico no es un proceso lineal, sino un modelo mental que nos ayuda a pensar en nuestras interfaces de usuario como un todo cohesivo y una colección de partes al mismo tiempo. Cada una de las cinco etapas juega un papel clave en la jerarquía de nuestros sistemas de diseño de interfaz [4].

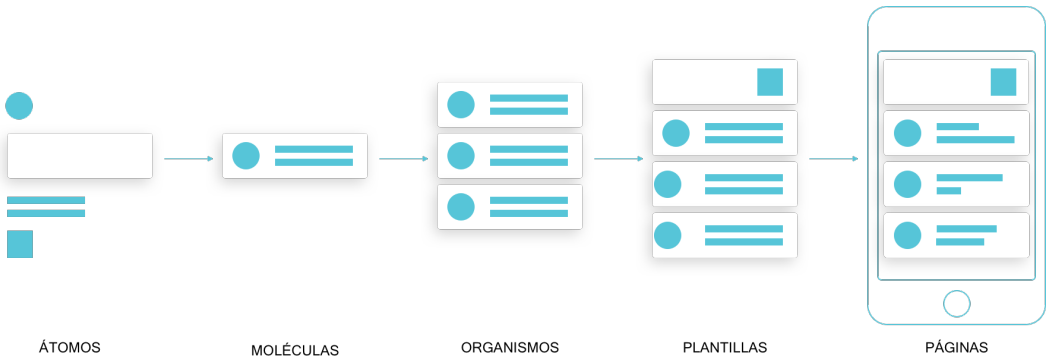


Figura 3.8: Metodología de diseño atómico. (Fuente: Elaboración Brad Frost 2016)

3.3.6. Estructura de la aplicación web

La interfaz de usuario juega un papel muy importante en el aumento de la usabilidad de una aplicación, dado que la IU ofrece al usuario una vista abstracta de todo el sistema, el éxito del sistema depende en gran medida de ello. Por lo tanto, el diseño de la interfaz de usuario debe tener la importancia adecuada en el proceso del ciclo de vida del diseño del sistema.

Módulo de inicio de sesión

Uno de los desafíos es cómo implementar un esquema de autenticación y autorización flexible, seguro y eficiente. Parece confuso diferenciar entre autenticación y autorización. De hecho, es muy simple.

- Autenticación: se refiere a verificar 'quién es usted', por lo que debe usar el nombre de usuario y la contraseña para la autenticación.
- Autorización: se refiere a lo que 'puede hacer', por ejemplo, acceder, editar o eliminar datos a algunos documentos, esto sucede después de la verificación.

Firebase Authentication proporciona servicios de back-end, SDK fáciles de usar y bibliotecas de interfaz de usuario para autenticar a los usuarios en la aplicación. En el código ejemplo 3.16 se muestran los métodos necesarios para el manejo de sesiones del proyecto.

Código Ejemplo 3.16: Fragmento de código del manejo de sesión de usuario

```
import * as firebase from 'firebase';
import Cookies from 'js-cookie'; // Módulo para el manejo de cookies.

const auth = firebase.auth(); // Inicializa Firebase Authentication.
```

```
// Agrega una cookie con el token de usuario.
const setAppCookie = () => (
  auth.currentUser.getIdToken().then((token) => {
    Cookies.set('token', token, { ... });
  })
);

// Remueve la cookie en logout
const unsetAppCookie = () => Cookies.remove('token', { ... });

// Detecta cambios en el usuario de Firebase Auth
const onAuthChange = () => (
  dispatch => auth.onAuthStateChanged((user) => {
    if (user) { // Si el usuario esta autenticado.
      setAppCookie(); // Se inicializan las conexiones
      initFirestoreConnections(dispatch);
      socketIO = initSocketIO(user.uid, dispatch);
      dispatch(userLogged(user.user)); // Guarda el usuario en el estado
    } else { // Si el usuario no esta autenticado
      unsetAppCookie(); // Cerramos las conexiones
      closeSocketIO();
      socketIO = null;
      dispatch(userLogged()); // Elimina el usuario del estado
    }
  })
);
// Inicio de sesion con correo y contraseña.
const login = (mail, pass) => auth.signInWithEmailAndPassword(mail, pass);

// Salida de sesion.
const logout = () => auth.signOut();
```

Gracias a la simplicidad y efectividad de los servicios de Firebase, este proceso es utilizado por muchas aplicaciones y servicios web en la actualidad.

Evaluación de la interfaz de usuario

El objetivo del proyecto es señalar los problemas y desafíos que surgen del uso de una pantalla táctil como dispositivo de entrada en una aplicación web. Para lograr esto, se desarrolló un prototipo, basado en pautas teóricas. El prototipo fue evaluado en sesiones informales donde los usuarios lo probaron y hablaron sobre cómo lo percibieron. Se alentó a los sujetos a sugerir soluciones alternativas y criticar las soluciones que encontraron negativas.

El prototipo inicial sufrió muy pocos cambios, siendo principalmente por motivos de funcionalidad y no estéticos. Todos los usuarios que sometieron a prueba la aplicación acordaron que el diseño era fácil de entender y se podían acostumbrar a el rápidamente. El resultado de la evaluación demostró que la interfaz responsiva funciona adecuadamente.

3.4. Configuración para producción

Hacer que las aplicaciones de Node.js estén listas para producción es probablemente el tema más oculto y omitido en la literatura de Node.js, pero es uno de los más importantes [8]. Se requiere de un método estandarizado para que la efectividad del despliegue en el servidor de producción sea optima.

Requisitos del servidor

- Servidor CentOS 7 con Node.js y Git instalado.
- Al menos 512 Mb de RAM y 15 Gb de espacio libre en disco.
- Acceso de usuario root a través de SSH.
- Un nombre de dominio apuntado a la dirección IP del servidor (rosesland.app)

Servidor Privado Virtual

Esta tecnología permite ejecutar las instancias de múltiples servidores en un servidor host físico de forma segura. Es un tipo de servidor de alojamiento web donde el alojamiento generalmente se realiza dividiendo un servidor físico principal en diferentes servidores virtuales múltiples. Cada uno de los servidores del sistema obtiene su propia parte de los recursos en función de los requisitos del cliente.

Este proyecto utiliza un servidor privado virtual (VPS) alojado en Digital Ocean, en donde debe decidir qué sistema operativo se desea, configurar el acceso SSH, crear un firewall, instalar los distintos lenguajes que necesita, etcétera. Para crear el VPS para el proyecto se accede a www.digitalocean.com y se crea una cuenta en Digital Ocean. Es necesario verificar la cuenta con una dirección de correo electrónico antes de poder crear un proyecto. Una vez verificada la dirección de correo electrónico, deberá un método de pago para el servicio. Al finalizar el registro y el modo de financiamiento se puede

comenzar a configurar un servidor o Droplet. En Digital Ocean un Droplet es una máquina virtual simple y escalable. Con un costo de solo 5 dolares al mes, el plan básico es lo suficiente para mantener el proyecto en operación.

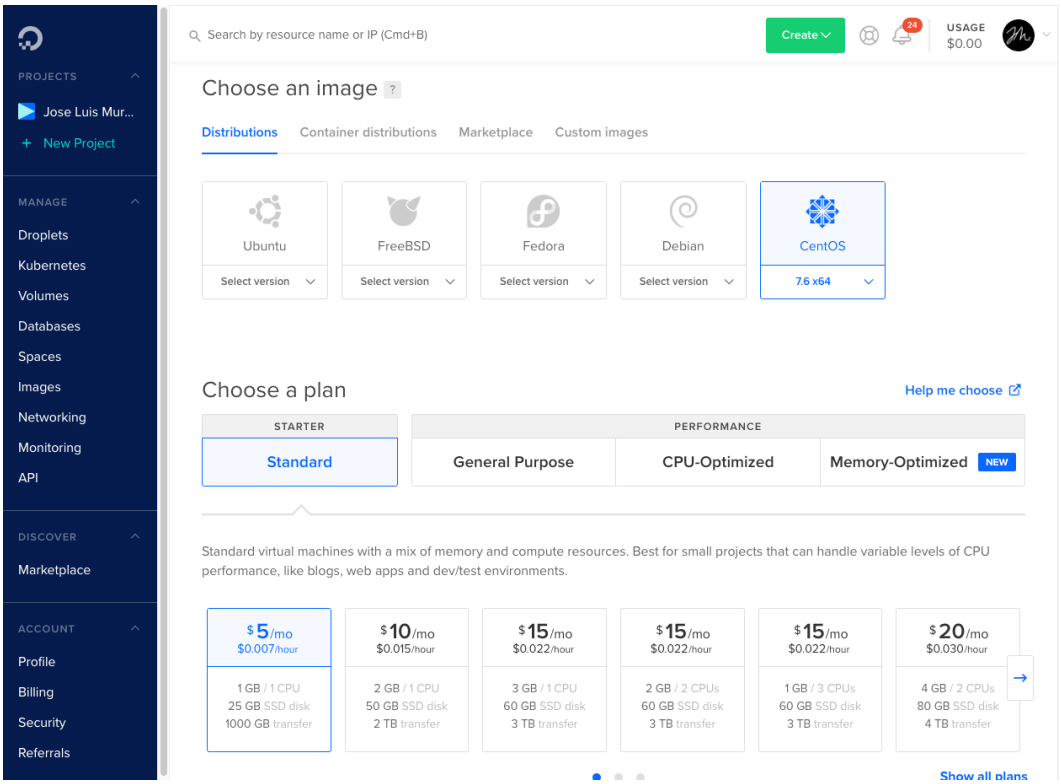


Figura 3.9: Configuración básica del servidor virtual privado. (Fuente: Digital Ocean)

Una vez que el servidor es creado, se recibe un correo electrónico de Digital Ocean. Este correo electrónico contiene los detalles de inicio de sesión para el servidor, en donde podremos conectarnos mediante SSH.

3.4.1. Configurar despliegue automático con Git

Al utilizar Git, el flujo de trabajo generalmente se dirige solo al control de versiones. Se tiene un repositorio local donde se trabaja y un repositorio remoto donde se mantiene todo sincronizado y se puede trabajar con un equipo y diferentes máquinas. Pero también es posible usar Git para mover una aplicación a producción [14].

Configuración de repositorios

Para poder empujar cambios a nuestro repositorio remoto se debe establecer la siguiente estructura:

- Directorio publico del servidor: /var/www/rosesland.app
- Directorio del repositorio del servidor: /var/repo/site.git

Se inicia sesión en el VPS desde la consola de comandos y se ingresa lo siguiente:

```
cd /var
mkdir repo && cd repo
mkdir site.git && cd site.git
git init --bare
```

`--bare` significa que la carpeta no tendrá archivos fuente, solo el control de versiones.

Git Hooks

Los repositorios de Git tienen una carpeta llamada `hooks`. Esta carpeta contiene algunos archivos de muestra para conectar y realizar acciones personalizadas. La documentación de Git define tres posibles enlaces de servidor:

pre-recepción, *post-recepción* y *actualización*. Para este proyecto se necesita un '*hook*' para post-recepción, que se ejecuta cuando un 'envío' está completamente terminado.

En el repositorio se encuentran algunos archivos y carpetas, incluida la carpeta `hooks`. Se accede a ella mediante el siguiente comando.

```
cd hooks
```

Se crea el archivo post-recepción escribiendo:

```
cat > post-receive
```

Al ejecutar este comando, se muestra una línea en blanco que indica que todo lo que escriba se guardará en este archivo. Se debe agregar lo siguiente:

```
#!/bin/sh
git -work-tree=/var/www/rosesland.app -git-dir=/var/repo/site.git checkout -f
```

Al terminar, se presiona 'control-d' para guardar. Para ejecutar el archivo, se necesita establecer los permisos adecuados usando:

```
chmod +x post-receive
```

El archivo post-recepción se examina cada vez que se completa un envío y coloca los archivos en `/var/www/rosesland.app`, esto permite actualizaciones directas del proyecto facilitando el mantenimiento de codigo y evitando la tarea de tener que conectarse al servidor para cargar los archivos manualmente.

En el repositorio local se necesita configurar la ruta del repositorio remoto. Se le dice Git que agregue un control remoto llamado 'live':

```
git remote add live ssh://root@rosesland.app/var/repo/site.git
```

Con esto es posible empujar cambios en el repositorio al servidor remoto usando el alias 'live':

```
git push live master
```

Con esto se le dice a Git que empuje al remoto 'live' en la rama 'master'.

3.4.2. Despliegue Node.js en un entorno de producción

El término servidor puede ser un poco confuso para las personas nuevas en el tema porque puede referirse tanto al *hardware* (computadoras físicas que albergan todos los archivos requeridos por los sitios web) como al *software* (programa que permite a los usuarios acceder a esos archivos en la web). El *hardware* se encuentra alojado en un servidor virtual privado, por tal motivo esta sección se centra en el lado del software.

Servidor proxy inverso

Una vez transferido el código del proyecto al servidor y configurar un entorno para la aplicación, es momento de configurar un servidor proxy inverso. Un servidor proxy es un servidor intermedio o intermediario que reenvía solicitudes de contenido de varios clientes a diferentes servidores a través de Internet. Un servidor proxy inverso es un tipo de servidor proxy que generalmente se encuentra detrás del firewall en una red privada y dirige las solicitudes de los clientes al servidor apropiado. Un proxy inverso proporciona un nivel adicional de abstracción y control para garantizar el flujo fluido del tráfico de red entre clientes y servidores.

Configuración nginx y firewall-cmd

Nginx (Engine X) es una de las mejores soluciones para configurar un servidor proxy inverso, con el será posible entregar contenido de forma rápida,

confiable y segura. Hará que la aplicación sea accesible desde un navegador, y en caso de ejecutar varios sitios desde el mismo servidor, también podría funcionar como equilibrador de carga.

Después de iniciar sesión como usuario root, se agrega el repositorio CentOS 7 *EPEL* y se instala nginx con los siguientes comandos:

```
yum install epel-release
yum install nginx
```

Se presiona 'y' dos veces para finalizar la instalación. Para inicializar el servicio nginx y este inicie con el arranque del servidor se introducen los siguientes comandos:

```
sudo systemctl enable nginx
sudo systemctl start nginx
```

A continuación se instala firewall-cmd, el front-end de línea de comandos para firewalld (*daemon* firewalld), para CentOS. Admite IPv4 e IPv6, zonas de firewall, puentes y conjuntos de ip, registra paquetes denegados, carga automáticamente módulos de *kernel*, entre otras cosas más. Se instala con el siguiente comando:

```
yum install firewalld
```

Ahora, se habilita para que se inicie automáticamente en el arranque del sistema y se observa su estado:

```
systemctl start firewalld
systemctl enable firewalld
systemctl status firewalld
```

En nginx para CentOS se necesita crear carpetas para sitios disponibles y habilitados. Se crean con los siguientes comandos:

```
mkdir /etc/nginx/sites-available
```

```
mkdir /etc/nginx/sites-enabled
```

Después se edita el archivo de configuración global de nginx:

```
nano /etc/nginx/nginx.conf
```

Se identifica la linea:

```
include /etc/nginx/conf.d/*.conf;
```

Se inserta lo siguiente:

```
include /etc/nginx/sites-enabled/*;
server_names_hash_bucket_size 64;
```

Código Ejemplo 3.17: Fragmento de configuración nginx

```
http {
  include       /etc/nginx/mime.types;
  default_type  application/octet-stream;

  log_format  main  '$remote_addr - $remote_user [
     $time_local] "$request" '
                    '$status $body_bytes_sent "
                       $http_referer" '
                    '"$http_user_agent" "
                       $http_x_forwarded_for"';
  access_log  /var/log/nginx/access.log  main;
  sendfile        on;
  keepalive_timeout  65;

  #Se agrega debajo de esta linea
  include /etc/nginx/conf.d/*.conf;
  include /etc/nginx/sites-enabled/*;
  server_names_hash_bucket_size 64;
}
```

Para finalizar se procede a utilizar el modulo npm llamado 'nginx-generator' para generar los archivos que le indican a nginx que actúe como un proxy inverso. En la linea de comandos se escribe lo siguiente:

```
nginx-generator \
 --name rosesland \
 --domain http://rosesland.app \
 --type proxy \
 --var host=localhost \
 --var port=3000 \
 /etc/nginx/sites-enabled/rosesland.app.conf
```

Este comando crea un archivo llamado `site_nginx` y lo coloca dentro del directorio `/etc/nginx/sites-enabled/`. El archivo se puede observar con el siguiente comando:

```
sudo nano /etc/nginx/sites-enabled/rosesland.app.conf
```

Por ultimo se reinicia nginx:

```
systemctl restart nginx
```

Configuración del administrador de procesos PM2

PM2 es un administrador de procesos de producción para aplicaciones basadas en Node.js. Con PM2, se pueden monitorear aplicaciones, su memoria y uso de CPU. También proporciona comandos fáciles para detener/iniciar/reiniciar todas las aplicaciones o aplicaciones individuales. Una vez que la aplicación se inicia a través de PM2, siempre se ejecutará después de que el sistema se bloquee o reinicie.

PM2 se instala mediante npm con el siguiente comando:

```
npm install pm2 -g
```

Una vez instalado sera posible ejecutar la aplicación con PM2:

```
pm2 start /var/www/rosesland.app/index.js --name rosesland
```

```
[root@rosesland-host rosesland.app]# pm2 list
```

App name	id	version	mode	pid	status	restart	uptime	cpu	mem	user	watching
rosesland	0	0.34.0	fork	30129	online	28	53D	0.3%	37.9 MB	root	disabled

```
Use `pm2 show <id|name>` to get more details about an app
```

Figura 3.10: Lista de procesos en pm2. (Fuente: Elaboración propia)

Asegurar el sitio mediante HTTPS

Con un nombre de dominio y registros DNS configurados correctamente para que apunten al VPS, se puede usar Certbot para generar certificados. Certbot es una herramienta de software gratuita y de código abierto para usar automáticamente certificados seguros en sitios web administrados manualmente para habilitar HTTPS [2].

Certbot está empaquetado en los Paquetes Adicionales para Enterprise Linux o EPEL (Extra Packages for Enterprise Linux). Para habilitar el repositorio adicional, el canal opcional e instalar EPEL, se emiten los siguientes comandos:

```
yum -y install yum-utils
yum-config-manager -enable \
rhui-REGION-rhel-server-extras rhui-REGION-rhel-server-optional
sudo yum install certbot python2-certbot-nginx
```

Se presiona 'y' cuando es requerido y al finalizar ejecutamos certbot:

```
certbot -nginx
```

Cuando se instalan certificados por primera vez, Certbot pide una dirección de correo electrónico de emergencia, luego varias preguntas menos importantes y,

finalmente, ¿desea redirigir todo el tráfico HTTP a HTTPS? Se selecciona la opción 2 para confirmar esta redirección y concluir con la configuración del despliegue de la aplicación.

En este momento se cuenta con una aplicación Node.js ejecutándose como un servicio en segundo plano en un entorno de producción, utilizando el protocolo HTTPS por seguridad y nginx como proxy inverso.

Capítulo 4

Conclusiones

Los procedimientos establecidos al crear este proyecto sirven como base para elaborar otras aplicaciones, su código es escalable y permite integrar un gran numero de servicios y *frameworks*, por lo que su limitación solo estará marcada por las necesidades de los usuarios.

Bibliografía

[1] Ethan Brown. *Web Development with Node and Express*. 2014. URL: http://www.vanmeegern.de/fileadmin/user_upload/PDF/Web_Development_with_Node_Express.pdf.

[2] *Certbot*. 2019. URL: https://certbot.eff.org/about/.

[3] Firebase. *Transacciones y escrituras en lotes*. 2019. URL: https://firebase.google.com/docs/firestore/manage-data/transactions.

[4] Brad Frost. *Atomic Design*. 2016. URL: http://www.softouch.on.ca/kb/data/Atomic%20Design.pdf.

[5] GoalKicker. *Node.js - Notes for Professionals*. URL: https://goalkicker.com/NodeJSBook/.

[6] Fionn Kelleher. *Understanding Socket.IO*. 2014. URL: https://nodesource.com/blog/understanding-socketio/.

[7] 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.

[8] Azat Mardan. "Getting Node.js Apps Production Ready". En: jul. de 2014, págs. 215-241. ISBN: 978-1-4302-6595-5. DOI: 10.1007/978-1-4302-6596-2_10.

[9] Suresh Mukhiya, Tao Wei y James Lee. *Redux Quick Start Guide*. Feb. de 2019.

[10] *Node.js web application framework*. URL: http://expressjs.com/.

[11] Jugnu Gaur Ochin. *Cross Browser Incompatibility: Reasons and Solutions*. Jul. de 2011. URL: http://www.airccse.org/journal/ijsea/papers/0711ijsea05.pdf.

[12] Stoyan Stefanov. *React: Up and Running*. 2016. URL: http://www.phpied.com/files/react/book/reactbook1-4.pdf.

[13] Srinivasan Subramanian. *Why MERN?* 2017. URL: https://www.apress.com/gp/blog/all-blog-posts/why-mern/12056000.

[14] Caio Vaccaro. *How To Set Up Automatic Deployment with Git with a VPS*. 2013. URL: https://www.digitalocean.com/community/tutorials/how-to-set-up-automatic-deployment-with-git-with-a-vps.