Integrantes

Álvaro Betancourt Guaba

Luisa Fernanda Quintero Ramírez

Jesús Daniel Molina Emiliani

Grupo 4

Taller 2

Asignatura

Arquitectura de software

Profesor

Andrés Sánchez Martín

MVC-Rest

Contenido

DEFINICIÓN, HISTORIA Y EVOLUCIÓN

MVC

Definición

El **Modelo Vista Controlador (MVC)** es un patrón arquitectónico que separa una aplicación en tres componentes principales: **Modelo, Vista y Controlador.** El **Modelo** gestiona los datos de la aplicación, la **Vista** se encarga de la representación visual, y el **Controlador** actúa como un intermediario que maneja la lógica y coordina las interacciones entre el modelo y la vista. Esta separación promueve una mayor modularidad y facilita el mantenimiento y escalabilidad de las aplicaciones, permitiendo que cada componente sea modificado independientemente sin afectar a los demás (Gamma, Helm, Johnson & Vlissides, 1994).

Historia y Evolución del MVC

El concepto de MVC fue introducido por **Trygve Reenskaug** en 1979 mientras trabajaba en el laboratorio de Xerox PARC para el lenguaje de programación **Smalltalk-80.** Su objetivo era crear una manera de organizar las interfaces de usuario de manera que fuesen fáciles de gestionar y más modulares (Reenskaug, 1979). Smalltalk utilizaba el patrón MVC como una solución a la creciente complejidad en la creación de interfaces gráficas, y se popularizó gracias a su éxito en la construcción de aplicaciones interactivas.

A lo largo de las décadas, el patrón MVC ha evolucionado y ha sido adoptado ampliamente en diferentes plataformas de desarrollo. Durante los años 90 y 2000, MVC ganó tracción en el desarrollo web, siendo utilizado en frameworks como **Ruby on Rails, Django** y **ASP.NET MVC,** debido a su capacidad para separar claramente la lógica del negocio de la interfaz de usuario, lo que permitió una mayor flexibilidad y mantenibilidad de las aplicaciones (Fowler, 2003).

Hoy en día, el patrón ha dado lugar a variantes más específicas, como **Model-View-ViewModel (MVVM)** y **Model-View-Presenter (MVP),** que se adaptan a necesidades específicas de plataformas móviles y web modernas, especialmente en la construcción de aplicaciones con interfaces dinámicas y ricas en interactividad (Fowler, 2003).

Características

1. **Separación de responsabilidades**: El patrón divide la aplicación en tres componentes principales:
    - **Modelo**: Gestiona la lógica de los datos y el acceso a la base de datos.

    - **Vista**: Representa la interfaz de usuario, mostrando los datos al usuario y permitiendo la interacción.
    - **Controlador**: Actúa como intermediario, recibiendo las solicitudes de la vista, procesando la lógica en el modelo y devolviendo la vista adecuada.
2. **Modularidad**: Cada componente puede desarrollarse y mantenerse de manera independiente, lo que facilita el mantenimiento y la escalabilidad de la aplicación.
3. **Desacoplamiento**: El MVC promueve un bajo acoplamiento entre la interfaz de usuario y la lógica de negocio, permitiendo que las vistas se modifiquen sin afectar a la lógica del negocio.
4. **Reusabilidad**: Los componentes del sistema (especialmente el modelo y las vistas) pueden ser reutilizados en otras partes de la aplicación o en diferentes proyectos.
5. **Soporte para múltiples vistas**: El mismo modelo puede ser utilizado por diferentes vistas, lo que permite la representación de los datos en distintos formatos.

Ventajas y Desventajas del MVC

Ventajas

- **Separación clara de responsabilidades**: Al dividir la aplicación en tres componentes distintos, facilita la organización del código y la separación entre lógica de negocio, presentación y control.
- **Escalabilidad**: MVC es adecuado para aplicaciones complejas, ya que permite añadir nuevas vistas y controladores sin afectar el modelo o la lógica subyacente.
- **Facilidad para el mantenimiento**: La modularidad del patrón MVC facilita la actualización o reemplazo de cualquiera de los componentes de la aplicación sin afectar a los demás.
- **Soporte para desarrollo en equipo**: Diferentes equipos pueden trabajar simultáneamente en los tres componentes (modelo, vista y controlador), lo que acelera el desarrollo.
- **Compatibilidad con arquitecturas RESTful**: El patrón MVC es una estructura natural para desarrollar APIs RESTful, ya que sigue un enfoque lógico de gestión de datos, representación y control.

Desventajas

1. **Curva de aprendizaje pronunciada**: Para los desarrolladores sin experiencia en patrones arquitectónicos, MVC puede ser complejo de entender y aplicar correctamente.
2. **Sobrecarga en aplicaciones pequeñas**: Para proyectos pequeños o sencillos, la separación de MVC puede resultar excesiva, aumentando la complejidad innecesariamente.
3. **Comunicación excesiva entre componentes**: En algunos casos, puede ser necesario un flujo constante de información entre los tres componentes, lo que puede introducir cierta sobrecarga en el rendimiento.
4. **Mantenimiento difícil en aplicaciones muy grandes**: Aunque MVC es ideal para modularidad, en sistemas extremadamente grandes y mal estructurados, la relación entre componentes puede volverse complicada y difícil de manejar.

Casos de Uso

- **Desarrollo de aplicaciones web**: El patrón MVC es ampliamente utilizado en el desarrollo de aplicaciones web, especialmente cuando se necesita gestionar interfaces de usuario complejas y dinámicas. Por ejemplo, plataformas como tiendas en línea o sistemas de gestión de contenido (CMS) utilizan MVC para manejar la interacción entre el usuario y los datos.
- **Aplicaciones empresariales**: En grandes empresas, donde la lógica de negocio y la presentación de los datos deben mantenerse separadas, MVC es ideal para construir sistemas de planificación de recursos empresariales (ERP) o gestión de relaciones con clientes (CRM).
- **Aplicaciones de escritorio y móviles**: Aunque es más popular en la web, MVC también se utiliza en el desarrollo de software de escritorio y aplicaciones móviles. Plataformas como iOS y Android permiten la implementación de MVC para organizar la lógica de las interfaces de usuario.

Casos de Aplicación

- **Ruby on Rails**: Es uno de los frameworks más populares que utiliza MVC para el desarrollo web. Permite que los desarrolladores creen aplicaciones web rápidamente siguiendo el patrón MVC.
- **Spring Framework**: En el entorno de Java, el Spring Framework, y más concretamente Spring Boot, facilita el desarrollo de aplicaciones web basadas en MVC. Empresas como Netflix y eBay han utilizado Spring Boot para desarrollar microservicios siguiendo este patrón.
- **ASP.NET MVC**: Microsoft también implementa MVC en su plataforma ASP.NET, permitiendo a los desarrolladores crear aplicaciones web escalables y modulares. Muchos sistemas empresariales, como CRM o ERP, utilizan ASP.NET MVC para gestionar grandes volúmenes de datos y usuarios.

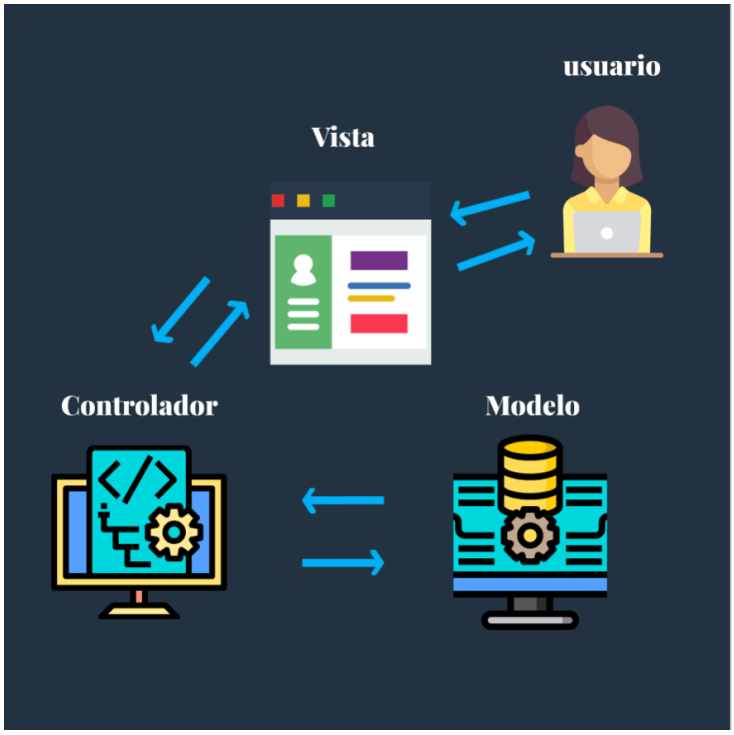


Figura 1. (Arquitectura de MVC)

La imagen anterior muestra una representación visual del patrón arquitectónico Modelo Vista Controlador (MVC). El usuario interactúa con la **Vista**, que es la interfaz gráfica que muestra la información y permite la interacción. La **Vista** a su vez se comunica con el **Controlador,** que es responsable de manejar la lógica de la aplicación, recibir las acciones del usuario y enviarlas al **Modelo.** El **Modelo** gestiona los datos, ya sea accediendo a una base de datos o realizando operaciones de procesamiento. El **Controlador** se encarga de actualizar el **Modelo** según sea necesario, y luego solicita a la **Vista** que se actualice para reflejar los cambios. Esta separación de responsabilidades en tres componentes permite una mejor organización y mantenimiento del código, además de facilitar el desarrollo escalable.

REST/JSON

Definición

REST (Representational State Transfer) y **JSON (JavaScript Object Notation)** son tecnologías clave en el desarrollo de aplicaciones modernas. **REST** es un estilo arquitectónico para el diseño de servicios web que permite que los recursos se definan y accedan a través de interfaces estándar, generalmente mediante el protocolo HTTP. **JSON**, por otro lado, es un

formato ligero de intercambio de datos, fácil de leer y escribir por humanos y sencillo de interpretar y generar por máquinas. JSON es ampliamente utilizado en REST para representar la información intercambiada entre cliente y servidor (Fielding, 2000; Crockford, 2006).

Historia y Evolución de REST

REST fue conceptualizado por **Roy Fielding** en su tesis doctoral en el año 2000 como un conjunto de principios arquitectónicos diseñados para aprovechar al máximo las capacidades de HTTP. A diferencia de otros modelos como SOAP, que son más complejos, REST utiliza métodos estándar de HTTP como **GET**, **POST**, **PUT** y **DELETE** para realizar operaciones CRUD (Create, Read, Update, Delete) en los recursos (Fielding, 2000). Su principal enfoque es la simplicidad y la escalabilidad, lo que lo ha convertido en el estilo más popular para el desarrollo de servicios web en la última década.

El diseño RESTful se centra en la idea de recursos, cada uno de los cuales tiene una representación que se puede acceder a través de una URL única. Los recursos se manipulan utilizando operaciones definidas por el protocolo HTTP, lo que hace que la arquitectura REST sea altamente compatible con la estructura de la web. En el contexto de la **arquitectura de software**, REST promueve el desacoplamiento entre cliente y servidor, lo que mejora la escalabilidad y flexibilidad del sistema. Además, su simplicidad facilita la integración de aplicaciones distribuidas (Pautasso, Zimmermann, & Leymann, 2008).

Historia y Evolución de JSON

JSON fue propuesto por **Douglas Crockford** a principios de los años 2000 como un formato de intercambio de datos derivado de JavaScript. Su objetivo era proporcionar una forma más sencilla y ligera de intercambiar datos entre sistemas comparado con XML, que era el formato más común en ese momento. Desde su introducción, JSON se ha convertido en el formato preferido para la comunicación entre sistemas, especialmente en servicios RESTful debido a su simplicidad y compatibilidad con la mayoría de los lenguajes de programación (Crockford, 2006).

A medida que las arquitecturas web y las APIs REST se hicieron más comunes, JSON se consolidó como el formato estándar para el intercambio de datos entre clientes (navegadores web o aplicaciones móviles) y servidores, gracias a su capacidad para representar objetos complejos de manera concisa y su facilidad para ser parseado (análisis sintáctico) por múltiples lenguajes (Zyp, 2007). JSON es ahora utilizado tanto en aplicaciones web como en sistemas distribuidos y es un componente esencial en arquitecturas orientadas a servicios y microservicios.

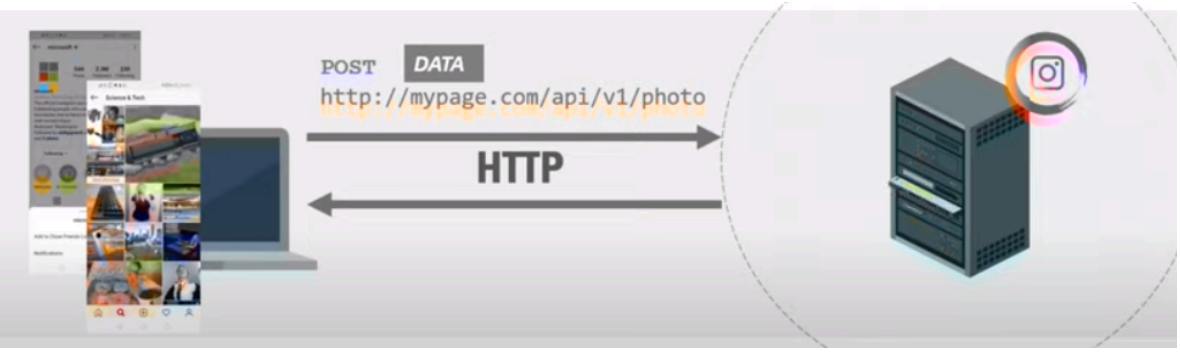


Fig 2 (Arquitectura REST API)

En el contexto del patrón arquitectónico **Modelo Vista Controlador (MVC**, los verbos HTTP juegan un papel esencial en la comunicación entre el cliente y el servidor. Los verbos más utilizados son **GET**, **POST**, **PUT/PATCH** y **DELETE**, y cada uno tiene un propósito específico dentro de las aplicaciones web que siguen la arquitectura MVC. **GET** se usa para solicitar recursos o información del servidor y devolverla al cliente, generalmente para cargar vistas o mostrar datos. **POST** se utiliza para crear nuevos recursos, como enviar formularios o agregar registros en una base de datos. **PUT** o **PATCH** permiten actualizar recursos existentes, mientras que **DELETE** elimina los recursos solicitados. Estas operaciones permiten que las aplicaciones MVC manejen las interacciones del cliente de manera clara y eficiente, enviando solicitudes desde la vista, procesándolas a través del controlador y gestionando los datos mediante el modelo.

La implementación de REST en una arquitectura basada en MVC se caracteriza por el intercambio de datos entre el cliente y el servidor como "cajas negras", es decir, cada parte no necesita conocer los detalles de implementación de la otra. Esto es posible porque las solicitudes contienen toda la información necesaria para que el servidor las procese, lo que permite que ambos trabajen de manera independiente, sin importar el lenguaje de programación o framework en que se implementen (Fielding, 2000). En este proceso, **JSON** se ha convertido en el formato preferido para el intercambio de datos entre el cliente y el servidor, debido a su simplicidad y compatibilidad con la mayoría de los lenguajes modernos.

Además, REST es un protocolo **stateless**, lo que significa que cada solicitud se maneja de manera independiente, sin necesidad de mantener información de peticiones anteriores. Sin embargo, el uso de **caching** en solicitudes **GET** permite que los recursos solicitados previamente se guarden, optimizando el rendimiento y reduciendo la carga del servidor al responder solicitudes repetitivas (Pautasso, Zimmermann, & Leymann, 2008). En las aplicaciones MVC, esta combinación de verbos HTTP y REST promueve una arquitectura eficiente, donde las interacciones entre la vista, el controlador y el modelo pueden llevarse a cabo de manera fluida y escalable.

Características de REST/JSON

- Stateless (sin estado): cada solicitud es independiente.

- Cacheable: las respuestas pueden ser cacheadas para mejorar la eficiencia.
- Formato JSON: formato ligero para la transferencia de datos entre cliente y servidor.
- Utiliza métodos HTTP estándar.

Ventajas y Desventajas de REST/JSON

Ventajas

- Independencia del lenguaje de programación en cliente y servidor.
- Ligero y fácil de integrar con múltiples plataformas.
- JSON es fácil de leer y de procesar por la mayoría de los lenguajes.

Desventajas

- No adecuado para transacciones complejas que requieren estado.
- Falta de estandarización de errores entre servicios REST.

Casos de Uso

- Creación de APIs para aplicaciones móviles y web.
- Integración de servicios de terceros con aplicaciones existentes.

Casos de Aplicación

- Facebook y Twitter utilizan APIs RESTful para permitir el acceso de aplicaciones de terceros.

SPRING BOOT

Definición

Spring Boot es un marco de desarrollo basado en **Spring**, diseñado para simplificar la creación de aplicaciones basadas en Java. Proporciona un enfoque preconfigurado que permite a los desarrolladores centrarse en la lógica de la aplicación sin preocuparse por la configuración manual de múltiples componentes del sistema. Spring Boot es particularmente útil para el desarrollo de aplicaciones web siguiendo el patrón **Modelo Vista Controlador (MVC)**, ya que proporciona una infraestructura completa para manejar las capas de presentación, lógica de negocio y persistencia de datos (Walls, 2016).

Historia y Evolución de Spring Boot

Spring Boot surgió como una respuesta a la complejidad que enfrentaban los desarrolladores al trabajar con el marco de desarrollo Spring tradicional. Mientras que Spring ofrece un potente conjunto de herramientas para construir aplicaciones empresariales, su flexibilidad y configuración extensiva a menudo dificultaban el desarrollo rápido de aplicaciones. En 2014, **Pivotal** lanzó **Spring Boot** como una solución para reducir la configuración manual y los archivos XML largos que eran necesarios en las versiones anteriores de Spring (Walls, 2016).

Spring Boot introduce una serie de características que facilitan la creación de aplicaciones "listas para producción", como un servidor web embebido (por ejemplo, Tomcat o Jetty), configuraciones automáticas, y la integración fluida con bibliotecas y servicios populares (Turnquist & Lomow, 2018). Además, Spring Boot se integra completamente con el ecosistema Spring, lo que permite a los desarrolladores aprovechar las capacidades de Spring Security, Spring Data, y Spring MVC, entre otros.

Desde su lanzamiento, Spring Boot ha ganado gran popularidad en la industria debido a su capacidad para desarrollar aplicaciones microservicios, aunque también se adapta perfectamente a arquitecturas tradicionales como el **Modelo Vista Controlador (MVC)**, donde permite gestionar de manera eficiente las interacciones entre el cliente y el servidor (Walls, 2016).

Características de Spring Boot

1. **Configuración automática (Auto-Configuration):** Spring Boot reduce la necesidad de configuraciones manuales al detectar automáticamente los componentes que necesita una aplicación. Con el soporte de configuraciones automáticas, Spring Boot puede ajustar las dependencias y los componentes adecuados según lo que esté presente en el classpath (Walls, 2016).

2. **Servidor web embebido:** Spring Boot proporciona servidores web embebidos como **Tomcat**, **Jetty**, y **Undertow**, lo que permite ejecutar aplicaciones como un archivo JAR sin necesidad de configurar un servidor externo (Turnquist & Lomow, 2018).

3. **Anotaciones simplificadas:** Spring Boot utiliza anotaciones como `@SpringBootApplication`, que encapsula otras tres anotaciones (`@Configuration`, `@EnableAutoConfiguration`, `@ComponentScan`), simplificando el código y la configuración.

4. **Manejo de dependencias (Spring Boot Starters):** Spring Boot incluye **starters**, que son paquetes que agrupan un conjunto de dependencias necesarias para ciertas funcionalidades, como el acceso a bases de datos, desarrollo web o mensajería. Esto facilita la gestión de dependencias en proyectos (Walls, 2016).

5. **Spring Boot CLI:** La **Command Line Interface (CLI)** permite a los desarrolladores escribir y ejecutar aplicaciones Spring Boot utilizando comandos de línea. Esto es útil para crear prototipos rápidos y pruebas.

6. **Motor de plantillas:** Spring Boot soporta diversos motores de plantillas como **Thymeleaf** y **Freemarker**, que se utilizan en la capa de **Vista** del patrón MVC para generar contenido HTML dinámico.

7. **Soporte para perfiles:** Spring Boot permite manejar diferentes configuraciones en función del entorno, utilizando perfiles como `dev`, `test`, o `prod`. Esto facilita la configuración de la aplicación en distintos entornos sin necesidad de cambiar el código fuente (Turnquist & Lomow, 2018).

8. **Monitorización y métricas:** Con **Spring Boot Actuator**, los desarrolladores pueden acceder a métricas, información de estado y endpoints de monitorización que permiten supervisar el comportamiento de la aplicación en tiempo real.

9. **Manejo simplificado de bases de datos:** Spring Boot incluye soporte para tecnologías de persistencia como **JPA/Hibernate** y bases de datos embebidas como **H2**. Esto facilita la interacción entre el **Modelo** y la base de datos en aplicaciones MVC.

10. **Desarrollo rápido de APIs REST:** Spring Boot ofrece un soporte robusto para la creación de APIs RESTful. Utilizando anotaciones como `@RestController` y mapeos como `@GetMapping`, `@PostMapping`, facilita la creación de servicios que pueden interactuar con clientes a través de HTTP.

11. **Extensa documentación y comunidad:** Spring Boot cuenta con una comunidad muy activa y una amplia documentación, lo que facilita la resolución de problemas y el aprendizaje continuo.

Ventajas y Desventajas de Spring Boot

Ventajas

- Configuración mínima y rápida.
- Integración directa con otros componentes de Spring.
- Servidores embebidos facilitan la portabilidad.
- Alta escalabilidad y modularidad.

Desventajas

- Puede ser pesado para aplicaciones pequeñas o simples.
- Curva de aprendizaje para desarrolladores sin experiencia en Spring.

Casos de Uso

- Aplicaciones web que requieren manejo eficiente de recursos y escalabilidad.
- Servicios RESTful para sistemas distribuidos.

Casos de Aplicación

- Empresas como Netflix y eBay han utilizado Spring Boot para implementar microservicios, aprovechando su capacidad para manejar grandes volúmenes de usuarios y peticiones.

ANGULAR/TYPESCRIPT

Definición

Angular es un framework de desarrollo web que facilita la creación de aplicaciones dinámicas, de una sola página (Single Page Applications - SPAs) y basadas en componentes. Desarrollado y mantenido por Google, Angular utiliza **TypeScript** como su lenguaje base, lo que le proporciona una sintaxis moderna y un tipado fuerte para aumentar la seguridad y eficiencia en el código. Angular simplifica la construcción de interfaces de usuario complejas al estructurar las aplicaciones en módulos y componentes reutilizables.

TypeScript es un superconjunto de JavaScript que añade tipado estático y otras características orientadas a objetos que hacen más eficiente y seguro el desarrollo de aplicaciones a gran escala. TypeScript transpila a JavaScript, por lo que es compatible con cualquier navegador o entorno donde se pueda ejecutar JavaScript.

Historia y Evolución de Angular

Angular fue lanzado originalmente en 2010 bajo el nombre de **AngularJS**. Esta primera versión fue revolucionaria, ya que introdujo el concepto de data binding bidireccional y facilitó la creación de aplicaciones web dinámicas sin necesidad de actualizar constantemente la página. AngularJS utilizaba JavaScript puro y rápidamente ganó popularidad debido a su simplicidad y capacidad para manejar el frontend de aplicaciones web modernas.

Sin embargo, con el tiempo, AngularJS empezó a mostrar limitaciones en cuanto a escalabilidad y rendimiento en aplicaciones más complejas. En 2016, Google lanzó **Angular 2**, que fue una reescritura completa del framework. Esta nueva versión ya no era compatible con AngularJS y adoptó **TypeScript** como su lenguaje principal, lo que mejoró la mantenibilidad y el rendimiento en proyectos grandes. A partir de esta versión, el framework comenzó a llamarse simplemente **Angular**.

Desde Angular 2, el framework ha evolucionado rápidamente, recibiendo actualizaciones constantes (hasta la versión actual, Angular 12) que han mejorado sus capacidades, su rendimiento y su integración con herramientas modernas de desarrollo como CLI (Command Line Interface) y bibliotecas de terceros. El uso de TypeScript ha permitido que Angular maneje aplicaciones más grandes y complejas con mayor facilidad, y el soporte de Google garantiza una evolución continua y mejoras en el framework.

Historia y Evolución de TypeScript

TypeScript, desarrollado por Microsoft, fue lanzado en 2012 como una alternativa tipada y mejor estructurada a JavaScript. A pesar de una adopción lenta al inicio, TypeScript ha ganado tracción, especialmente en frameworks como Angular, donde el tipado estático proporciona ventajas significativas en el desarrollo de aplicaciones empresariales a gran escala. Su evolución ha sido rápida, y hoy en día es uno de los lenguajes más populares para el desarrollo frontend, respaldado por la comunidad de código abierto y grandes empresas como Microsoft y Google.

En resumen, Angular, al pasar de su versión original en JavaScript a su versión moderna basada en TypeScript, ha logrado mantenerse como una de las tecnologías más robustas para el desarrollo frontend, permitiendo a los desarrolladores crear aplicaciones web ricas, interactivas y escalables.

Características de Angular/Typescript

- Componentes reutilizables.
- Binding bidireccional de datos (two-way data binding).
- Soporte para servicios HTTP para integrar con APIs RESTful.
- Integración con TypeScript para mayor seguridad y escalabilidad.
- Enrutamiento integrado para la navegación dentro de la aplicación.

Ventajas y Desventajas de Angular/Typescript

Ventajas

- Potente y flexible para el desarrollo de SPAs (Single Page Applications).
- TypeScript mejora la legibilidad y depuración.
- Buen soporte y comunidad activa.

Desventajas

- Curva de aprendizaje pronunciada debido a su complejidad.
- Las actualizaciones frecuentes pueden requerir refactorizaciones.

Casos de Uso

- Aplicaciones web interactivas con múltiples vistas.
- Aplicaciones empresariales donde la modularidad es clave.

Casos de Aplicación

- Google Ads utiliza Angular para construir interfaces ricas y altamente interactivas.

MongoDB

Definición

MongoDB es una base de datos **NoSQL** orientada a documentos que almacena datos en formato **BSON** (Binary JSON), una extensión binaria de JSON. A diferencia de las bases de datos relacionales tradicionales que usan tablas y filas, MongoDB organiza los datos en documentos flexibles, lo que facilita el manejo de grandes volúmenes de datos no estructurados o semi-estructurados. MongoDB es conocida por su capacidad para escalar horizontalmente y por su flexibilidad para cambiar esquemas sin afectar el rendimiento (Chodorow, 2013).

Historia y Evolución de MongoDB

MongoDB fue creado en 2007 por **Dwight Merriman** y **Eliot Horowitz** en la empresa **10gen** (que más tarde cambió su nombre a MongoDB Inc.). Originalmente, el equipo de 10gen estaba desarrollando una plataforma de cómputo en la nube, pero pronto se dieron cuenta de que el mayor desafío que enfrentaban era la necesidad de una base de datos escalable que pudiera manejar datos no estructurados de manera eficiente. Este desafío llevó a la creación de MongoDB, que fue lanzado como un proyecto de código abierto en 2009 (Banker, 2011).

- **2009:** MongoDB fue lanzado como un proyecto de código abierto, ganando popularidad

rápidamente debido a su capacidad para manejar datos no estructurados y su escalabilidad horizontal (Chodorow, 2013).

- **2010:** Se introdujo el **sharding**, que permite a MongoDB escalar horizontalmente distribuyendo datos entre múltiples servidores, lo que facilita el manejo de grandes volúmenes de datos sin pérdida de rendimiento (Banker, 2011).
- **2010:** Se implementaron los **Replica Sets**, una característica que permite la replicación automática de datos entre múltiples nodos para proporcionar redundancia y alta disponibilidad. Esto asegura que si un nodo falla, otro puede asumir su función, lo que mejora la tolerancia a fallos (Chodorow, 2013).
- **2016:** Se lanzó **MongoDB Atlas**, una solución de base de datos como servicio (DBaaS) en la nube, que permitió a los desarrolladores implementar y gestionar MongoDB en servicios de nube como AWS, Azure y Google Cloud sin la complejidad de gestionar la infraestructura subyacente (MongoDB Inc., 2016).
- **2018:** Con el lanzamiento de **MongoDB 4.0**, se introdujo el soporte para **transacciones ACID multi-documento**, lo que permitió a MongoDB competir más directamente con bases de datos relacionales en casos de uso que requerían transacciones consistentes (Chodorow, 2019).

MongoDB ha continuado evolucionando, añadiendo nuevas características y optimizaciones que lo convierten en una de las bases de datos NoSQL más robustas y utilizadas en el mercado. Desde su creación, ha sido adoptada ampliamente en diversas industrias, incluyendo tecnología, finanzas y comercio electrónico, por empresas como **Uber**, **eBay** y **Coinbase**, debido a su capacidad para manejar grandes cantidades de datos en tiempo real y su flexibilidad en la gestión de esquemas (Chodorow, 2019).

Características de MongoDB

- Almacena datos en documentos BSON, similar a JSON.
- Escalabilidad horizontal a través del sharding.
- Capacidad para manejar grandes volúmenes de datos no estructurados.
- Modelo flexible de esquemas, permitiendo cambios sin afectar la base de datos completa.

Ventajas y Desventajas de MongoDB

Ventajas

- Alta flexibilidad en la gestión de datos.
- Ideal para aplicaciones con grandes volúmenes de datos no estructurados.
- Fácil de escalar horizontalmente.

Desventajas

- Falta de soporte para transacciones ACID completas en comparación con bases de datos relacionales.
- Mayor consumo de memoria debido a la estructura de documentos.

Casos de Uso

- Aplicaciones que requieren manejar datos no estructurados o semi-estructurados.
- Aplicaciones en tiempo real, como plataformas de análisis de datos y redes sociales.

Casos de Aplicación

- Empresas como Uber y eBay utilizan MongoDB para manejar grandes volúmenes de datos no estructurados y proporcionar respuestas en tiempo real.

RELACIÓN ENTRE LOS TEMAS ASIGNADOS

En el contexto del patrón **Modelo Vista Controlador (MVC)**, los componentes asignados (Spring Boot con Java en el backend, Angular con TypeScript en el frontend, REST y JSON para la comunicación, y MongoDB como base de datos) proporcionan una arquitectura flexible y escalable para el desarrollo de aplicaciones web modernas. Esta combinación de tecnologías permite crear aplicaciones robustas con una clara separación de responsabilidades entre la lógica de negocio, la presentación y la persistencia de datos.

1. **Spring Boot y Java**: En el backend, Spring Boot ofrece un marco robusto para manejar la lógica de negocio y exponer APIs RESTful. Con el soporte de **REST y JSON**, el backend interactúa de manera eficiente con el frontend, permitiendo a los clientes acceder a los datos de la aplicación mediante peticiones HTTP.
2. **Angular y TypeScript**: Angular, en el frontend, gestiona la interfaz de usuario mediante componentes reutilizables que se comunican con el backend a través de las APIs REST expuestas por Spring Boot. Utilizando **HttpClient**, Angular consume y procesa datos en formato JSON provenientes del backend.
3. **MongoDB**: Como base de datos NoSQL, MongoDB ofrece una gran flexibilidad en el almacenamiento de datos no estructurados o semi-estructurados. A través de **Spring Data MongoDB**, Spring Boot interactúa con MongoDB, proporcionando una capa de persistencia que es fácil de gestionar y escalar horizontalmente.

Esta arquitectura permite que el frontend (Angular) y el backend (Spring Boot) se comuniquen de manera desacoplada mediante APIs RESTful, mientras MongoDB gestiona la persistencia de datos, garantizando escalabilidad y flexibilidad en la aplicación.

¿Qué tan común es el stack designado?

El stack **Spring Boot + Angular + REST/JSON + MongoDB** es **muy común** en el desarrollo de aplicaciones web, especialmente en proyectos que requieren una separación clara entre el frontend y el backend. Este stack es ampliamente utilizado en empresas tecnológicas y startups debido a varias razones:

1. **Spring Boot** es una de las plataformas más populares para el desarrollo de aplicaciones backend en Java, particularmente en entornos empresariales y microservicios.
2. **Angular** es uno de los frameworks más utilizados para la creación de Single Page Applications (SPAs), lo que lo convierte en una opción común para proyectos que requieren interfaces de usuario interactivas.
3. **REST/JSON** es el estándar más común para la creación de APIs que conectan clientes y servidores, permitiendo una arquitectura ligera y eficiente.
4. **MongoDB** es muy utilizado en aplicaciones que requieren alta escalabilidad y flexibilidad en el manejo de grandes volúmenes de datos no estructurados, especialmente en entornos de aplicaciones distribuidas.

Matriz de Análisis: Principios SOLID vs Temas

Principio SOLID	Spring Boot (Java)	Angular (TypeScript)	REST/JSON	MongoDB
S - Responsabilidad única	Controladores, servicios y repositorios tienen responsabilidades claras.	Componentes y servicios separados por funcionalidades.	Cada endpoint REST gestiona una única operación.	Documentos independientes que contienen datos específicos.
O - Abierto/Cerrado	Servicios y controladores extensibles sin modificar su implementación.	Los componentes pueden ser extendidos sin modificar su lógica base.	Las APIs REST pueden ser extendidas sin romper clientes existentes.	Nuevos campos en documentos no afectan los anteriores.

L - Sustitución de Liskov	Las interfaces permiten sustituir implementaciones sin romper el código.	Los servicios de Angular pueden ser reemplazados fácilmente.	Los recursos y métodos REST pueden cambiar sin afectar la lógica de cliente.	MongoDB permite cambiar la estructura del documento sin afectar el código.
I - Segregación de interfaces	Interfaces especializadas permiten manejar dependencias desacopladas.	Los servicios son divididos para evitar dependencias grandes.	Exposición de solo los métodos REST necesarios.	MongoDB permite obtener solo los datos necesarios sin exponer toda la colección.
D - Inversión de dependencias	Spring Boot utiliza inyección de dependencias para desacoplar componentes.	Angular usa inyección de dependencias para desacoplar servicios de componentes.	REST es independiente del cliente, promoviendo un desacoplamiento fuerte.	La abstracción a través de servicios de persistencia desacopla el acceso directo a los datos.

Matriz de Análisis: Atributos de Calidad vs Temas

Atributo de Calidad	Spring Boot (Java)	Angular (TypeScript)	REST/JSON	MongoDB

Escalabilidad	Escalabilidad horizontal mediante microservicios.	Modularidad de componentes facilita la escalabilidad.	Las APIs REST son fácilmente escalables.	MongoDB permite escalabilidad horizontal mediante sharding.
Rendimiento	Alto rendimiento mediante caché y concurrencia.	Alta eficiencia en la renderización del DOM.	JSON es ligero y rápido para la transferencia de datos.	MongoDB es altamente eficiente en la manipulación de grandes volúmenes de datos.
Seguridad	Integración con Spring Security.	Seguridad mediante JWT y autenticación OAuth.	API REST puede ser asegurada mediante tokens JWT.	Control de acceso basado en roles a nivel de base de datos.
Mantenibilidad	Mantenimiento simplificado por la modularidad de los servicios.	Componentes fácilmente mantenibles y actualizables.	REST es fácil de mantener y extender.	Cambios en los documentos no afectan el funcionamiento global.
Disponibilidad	Alta disponibilidad mediante mecanismos de failover.	Angular permite mantener el frontend disponible.	REST soporta la disponibilidad en aplicaciones distribuidas.	Alta disponibilidad mediante réplica de datos.

Matriz de Análisis: Tácticas vs Temas

Táctica	Spring Boot (Java)	Angular (TypeScript)	REST/JSON	MongoDB

Tolerancia a fallos	Redundancia mediante microservicios replicados.	Angular puede funcionar con fallback en componentes.	El servidor puede replicarse fácilmente para mayor disponibilidad.	Réplica de datos y failover para alta disponibilidad.
Modularidad	Los microservicios permiten una modularidad alta.	Angular es altamente modular mediante componentes.	Las APIs REST son modulares por entidad.	Documentos independientes facilitan la modularidad.
Seguridad	Integración con OAuth2 y autenticación mediante JWT.	Autenticación basada en OAuth2 y JWT.	Seguridad mediante HTTPS y tokens JWT.	Seguridad de acceso a nivel de documentos y colecciones.

Matriz de Análisis: Mercado Laboral vs Temas

Tecnología	Demanda en el mercado laboral
Spring Boot (Java)	Alta demanda en desarrollo backend y microservicios.
Angular (TypeScript)	Alta demanda para desarrollo frontend de SPAs y aplicaciones empresariales.
REST/JSON	Estándar de facto en APIs, alta demanda en integración de sistemas.
MongoDB	Popular en el manejo de grandes volúmenes de datos y aplicaciones distribuidas.

Matriz de Análisis: Patrones Laborales vs Temas

Patrones de Diseño	Spring Boot (Java)	Angular (TypeScript)	REST/JSON	MongoDB

Patrón MVC	Implementa el patrón MVC en el backend.	Se ajusta a la arquitectura MVC como la capa de vista.	Utiliza el patrón MVC para manejar las rutas.	Se integra con el modelo de datos del patrón MVC.
Inyección de dependencias	Utiliza DI mediante Spring Framework.	Inyección de dependencias para modularidad.	Facilita la independencia de cliente y servidor.	MongoDB abstrae el acceso a datos para desacoplar la lógica de persistencia.

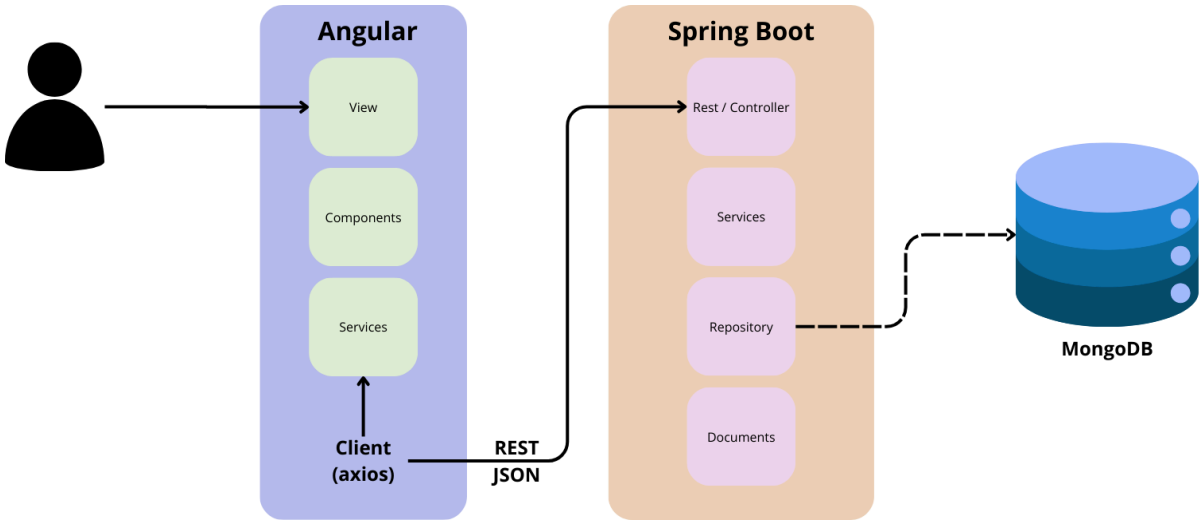


Figura 3 (*Diagrama de Arquitectura Angular - Spring Boot - MongoDB*)

Como se observa en la figura 3, la arquitectura de la aplicación web está dividida en dos partes principales: el **frontend**, gestionado por **Angular**, y el **backend**, controlado por **Spring Boot**, con **MongoDB** como base de datos. Cada componente juega un rol crucial en el manejo de la interfaz de usuario, la lógica de negocio y la persistencia de datos.

Lado del Cliente - Angular

En el **lado del cliente**, Angular se encarga de la **interfaz de usuario**. Los **templates** (vistas) en Angular definen la estructura visual que los usuarios ven e interactúan, mientras que los **componentes** controlan la lógica y comportamiento de la aplicación, manejando las interacciones del usuario con la interfaz.

Los **servicios (services)** son responsables de gestionar la comunicación con el backend. Utilizan la librería **HttpClient** de Angular (o en este caso, **Axios**) para enviar y recibir datos del servidor mediante solicitudes HTTP (GET, POST, PUT, DELETE). Estos servicios actúan como intermediarios entre los componentes de Angular y el servidor Spring Boot, permitiendo que la lógica de negocio se mantenga organizada y desacoplada de la vista. Todo intercambio de datos entre el frontend y el backend se realiza en formato **JSON**, un formato ligero que facilita la transferencia de información.

Lado del Servidor - Spring Boot

En el **lado del servidor**, **Spring Boot** actúa como el **backend** de la aplicación. Los **controladores REST** de Spring Boot son los encargados de recibir las solicitudes HTTP provenientes del cliente Angular, procesarlas y devolver una respuesta en formato JSON. Los **servicios** de Spring Boot contienen la lógica de negocio, realizando operaciones complejas sobre los datos, como cálculos, validaciones o transformaciones, antes de interactuar con la base de datos.

En lugar de una base de datos SQL como en otras arquitecturas, en esta aplicación, **MongoDB** es utilizado como base de datos **NoSQL**. Spring Boot utiliza **Spring Data MongoDB**, que proporciona una capa de abstracción para interactuar con MongoDB de manera eficiente. En lugar de utilizar consultas SQL tradicionales, **Spring Data MongoDB** maneja los datos en forma de **documentos BSON**, permitiendo realizar operaciones CRUD (crear, leer, actualizar, eliminar) sobre los datos de manera directa y flexible.

Los documentos almacenados en MongoDB representan las entidades de la aplicación, similares a las tablas de una base de datos relacional, pero con la flexibilidad que ofrecen las bases de datos NoSQL. Los **repositorios** en Spring Boot se encargan de gestionar estas operaciones sobre MongoDB, utilizando métodos como `save()`, `findById()`, `findAll()`, y `deleteById()` sin necesidad de escribir consultas manuales.

Ciclo de Interacción entre el Frontend y Backend

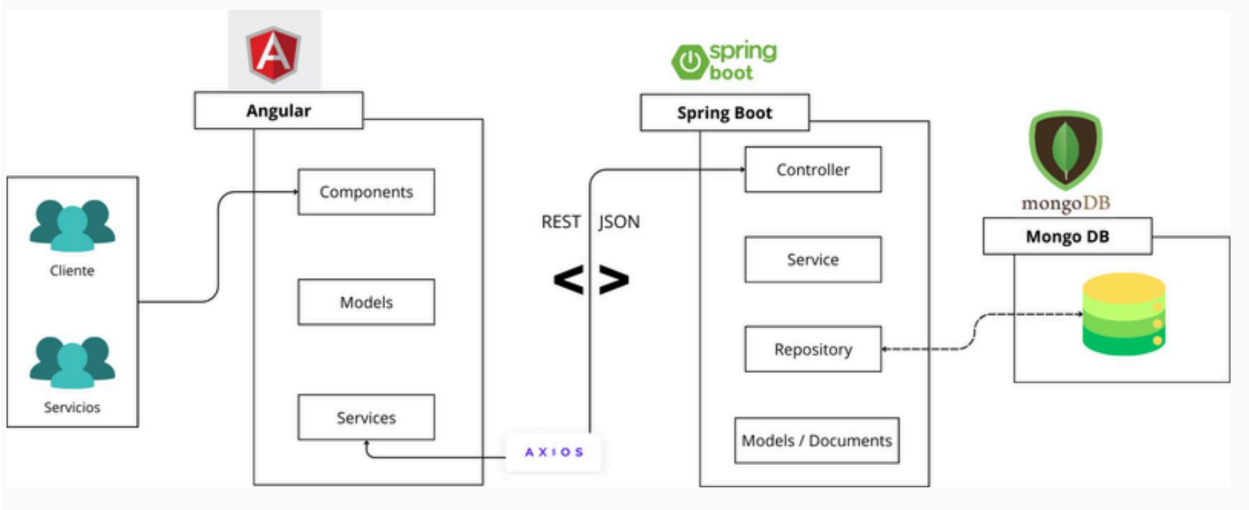
Cuando el usuario realiza una acción desde la **interfaz de Angular**, como por ejemplo enviar un formulario o solicitar datos, el **servicio** correspondiente en Angular utiliza **HttpClient** (o Axios) para enviar una **solicitud HTTP** al servidor **Spring Boot**. Esta solicitud se envía al **controlador REST** del backend, que recibe la solicitud y, a través de los **servicios de Spring**, procesa la lógica necesaria. Si se requiere interacción con los datos, el servicio de Spring utiliza **Spring Data MongoDB** para acceder a la base de datos y realizar las operaciones requeridas.

Una vez completadas las operaciones en el servidor (ya sea para obtener o modificar datos), el controlador REST devuelve una **respuesta en formato JSON** al cliente Angular. Angular recibe estos datos a través del servicio correspondiente, y los componentes se encargan de renderizar o actualizar la interfaz de usuario de acuerdo a la respuesta recibida, completando el ciclo de interacción.

Este flujo continuo entre **Angular** y **Spring Boot** a través de **APIs RESTful** garantiza una arquitectura modular, escalable y eficiente, donde el frontend y backend están completamente desacoplados, permitiendo un mantenimiento más sencillo y una mayor flexibilidad para escalar la aplicación en el futuro.

DIAGRAMAS

Diagrama de Arquitectura de Alto Nivel



C4MODEL

Diagrama de Contexto

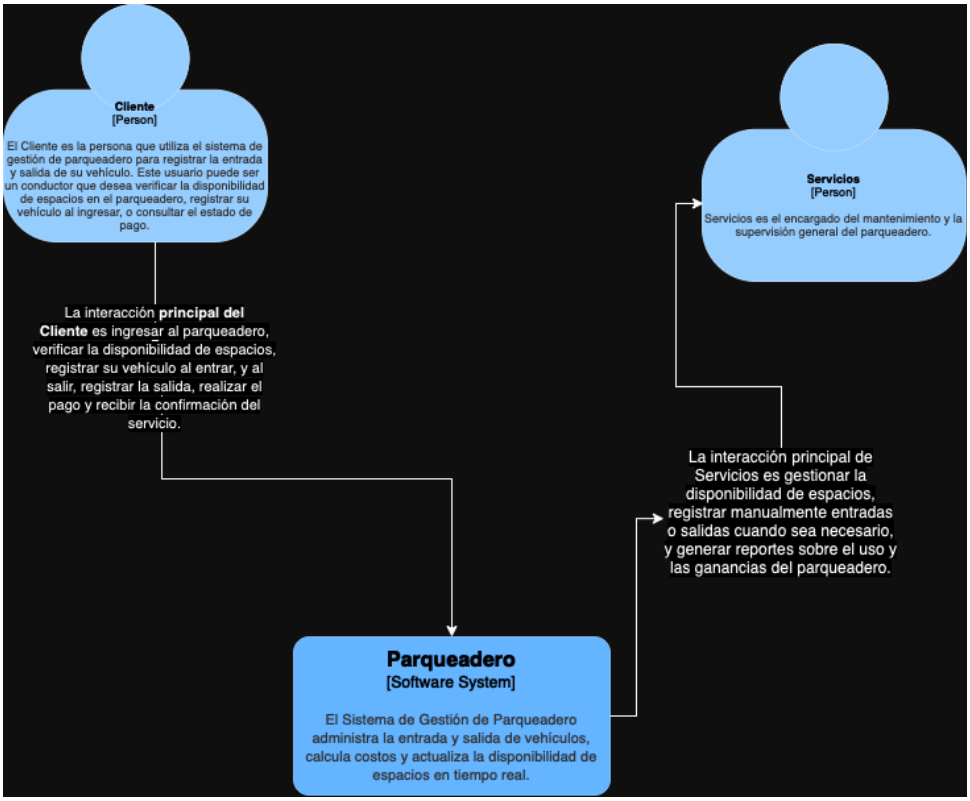


Diagrama de Contenedores

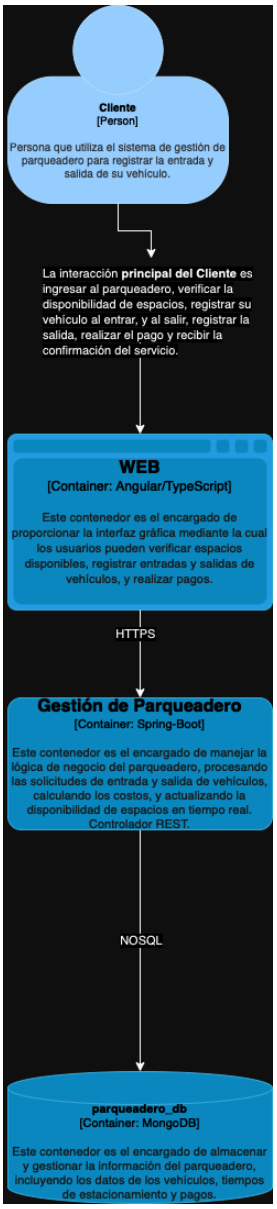


Diagrama de Componentes

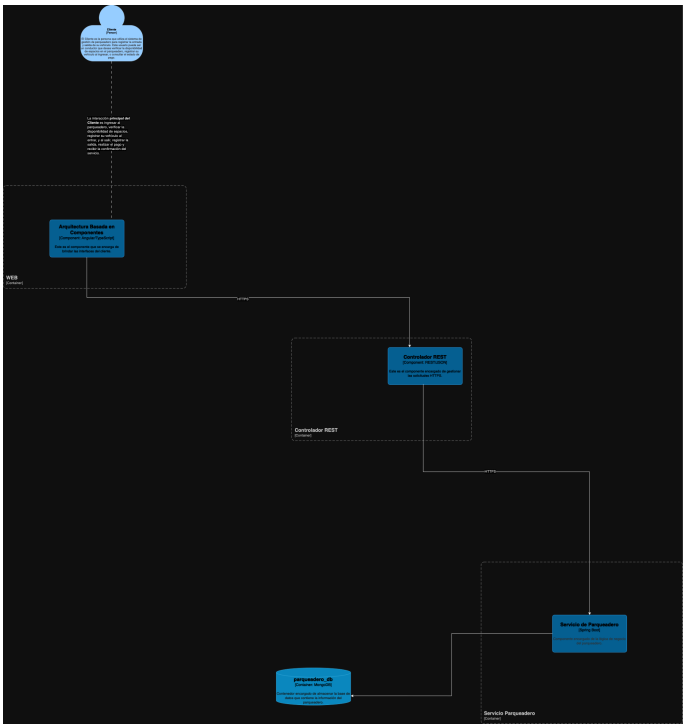


Diagrama de Despliegue

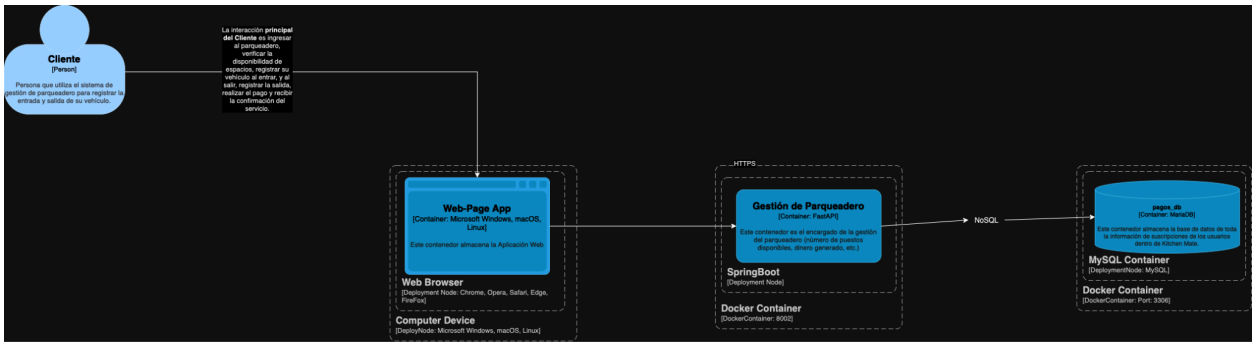
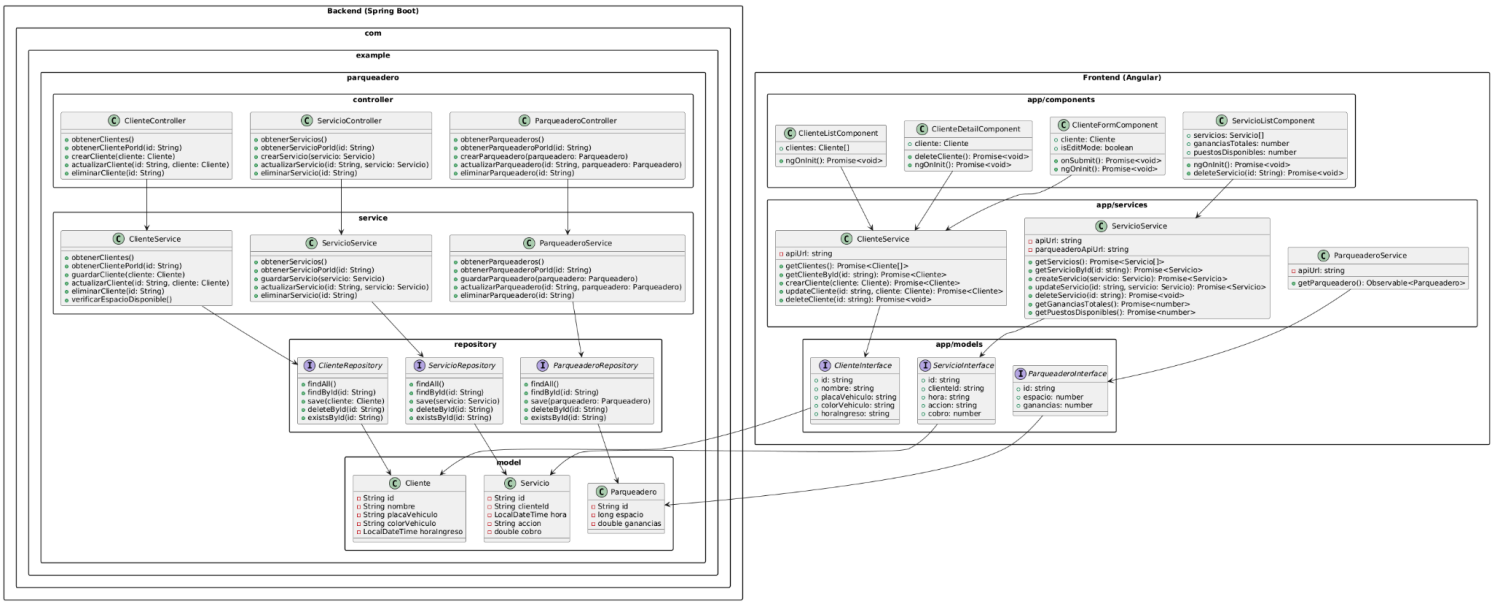


Diagrama de paquetes UML de cada componente



REFERENCIAS

- Fowler, M. (2003). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Reenskaug, T. (1979). *The Original MVC Reports*.
- Crockford, D. (2006). *The application/json Media Type for JavaScript Object Notation (JSON)*. IETF. https://www.ietf.org/rfc/rfc4627.txt
- Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (Doctoral dissertation, University of California, Irvine). https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
- Pautasso, C., Zimmermann, O., & Leymann, F. (2008). RESTful web services vs. "big" web services: making the right architectural decision. *Proceedings of the 17th international conference on World Wide Web*, 805-814.
- Zyp, K. (2007). *JSON: The Fat-Free Alternative to XML*. O'Reilly Media. https://www.oreilly.com
- Turnquist, G., & Lomow, C. (2018). *Learning Spring Boot 2.0: Simplify the development of lightning-fast applications based on microservices and reactive programming*. Packt Publishing Ltd.
- Walls, C. (2016). *Spring Boot in Action*. Manning Publications.
- Banker, K. (2011). *MongoDB in Action*. Manning Publications.
- Chodorow, K. (2013). *MongoDB: The Definitive Guide*. O'Reilly Media.
- Chodorow, K. (2019). *MongoDB 4.0: The Definitive Guide*. O'Reilly Media.
- MongoDB Inc. (2016). *Introducing MongoDB Atlas: The Database as a Service for MongoDB*.