CIENTIFICA
TechNoLogica

ISSN:3119-7906, Vol.1Núm.1(2026): Enero-Marzo

https://doi.org/10.17268/tech1.1.2026.02

Diseño e implementación de una arquitectura MVC full-stack para un sistema de reservas multinegocio

Design and implementation of a full-stack MVC architecture for a multi-business reservation system

Suárez López, Daniel[1].

[1] Escuela de Ingeniería Informática, Universidad Nacional de Trujillo, La Libertad, Perú.

RESUMEN

La gestión de reservas en entornos multinegocio presenta desafíos relacionados con la escalabilidad, la separación de responsabilidades y la coherencia entre la lógica de negocio, la persistencia de datos y la interfaz de usuario. Muchos sistemas existentes ofrecen soluciones rígidas o acopladas que dificultan su mantenimiento y evolución. En este contexto, el presente artículo describe el diseño e implementación de una arquitectura MVC full-stack aplicada a un sistema de reservas multinegocio orientado a un modelo SaaS. La propuesta integra una base de datos relacional gestionada mediante Prisma, un backend desarrollado con NestJS que actúa como controlador central de la lógica de negocio y la seguridad, y un frontend construido con Next.js y componentes reutilizables que representan la capa de presentación. La metodología empleada se basa en el diseño modular, la aplicación de patrones de diseño y la validación progresiva de los flujos del sistema. Como resultado, se obtiene una arquitectura funcional, escalable y preparada para su extensión, que facilita la gestión de usuarios, negocios, servicios y reservas en un entorno multinegocio. Las conclusiones evidencian que el enfoque MVC a nivel de sistema completo mejora la mantenibilidad y claridad estructural del software.

Palabras clave: Arquitectura MVC, sistemas de reservas, aplicaciones SaaS, desarrollo full-stack, patrones de diseño.

ABSTRACT

Managing reservations in multi-business environments presents challenges related to scalability, separation of res ponsibilities, and consistency between business logic, data persistence, and the user interface. Many existing systems provide rigid or tightly coupled solutions that hinder maintenance and evolution. In this context, this article describes the design and implementation of a full-stack MVC architecture applied to a multi-business booking system oriented to a SaaS model. The proposal integrates a relational database managed through Prisma, a backend developed with NestJS that acts as the central controller of business logic and security, and a frontend built with Next.js and reusable components that represent the presentation layer. The methodology is based on modular design, the application of design patterns, and the progressive validation of system workflows. As a result, a functional, scalable, and extensible architecture is obtained, facilitating the management of users, businesses, services, and bookings in a multi-business environment. The conclusions show that applying MVC at a system-wide level improves software maintainability and structural clarity.

Keywords: *MVC architecture, booking systems, SaaS applications, full-stack development, design patterns.*

1. INTRODUCCIÓN

La gestión de reservas es un problema recurrente en múltiples dominios, como servicios profesionales, negocios locales y plataformas digitales, donde la correcta organización de citas, disponibilidad y recursos resulta crítica para la eficiencia operativa. Tradicionalmente, muchos sistemas de reservas han sido desarrollados como soluciones monolíticas o altamente acopladas, lo que dificulta su mantenimiento, escalabilidad y adaptación a escenarios multinegocio [1], [2].

En el ámbito de la ingeniería de software, la arquitectura del sistema desempeña un papel fundamental en la calidad final de la aplicación. Diversos estudios destacan que una adecuada separación de responsabilidades facilita la mantenibilidad, la reutilización y la evolución del software a lo largo del tiempo [3], [4]. En este contexto, el patrón Modelo–Vista–Controlador (MVC) ha sido ampliamente adoptado como un enfoque arquitectónico que promueve la división clara entre la lógica de negocio, la presentación y el control de la aplicación [5], [6].

El crecimiento de las aplicaciones web modernas y de los modelos de negocio basados en Software como Servicio (SaaS) ha incrementado la necesidad de arquitecturas escalables y flexibles, capaces de soportar múltiples usuarios, organizaciones y flujos concurrentes. Las plataformas SaaS suelen requerir mecanismos de aislamiento lógico, control de acceso y escalabilidad horizontal, aspectos ampliamente discutidos en la literatura sobre computación en la nube [7], [8], [9].

Asimismo, el desarrollo de aplicaciones web full stack ha evolucionado hacia el uso de frameworks especializados tanto en el backend como en el frontend, los cuales permiten implementar arquitecturas modulares y orientadas a componentes. En este escenario, la aplicación de patrones de diseño, como los descritos por Gamma et al. y Fowler, contribuye a reducir el acoplamiento entre capas y a mejorar la claridad estructural del sistema [10], [11].

A pesar de estos avances, muchos sistemas de reservas existentes no integran de forma coherente los principios arquitectónicos con las necesidades específicas de entornos multinegocio, donde un mismo usuario puede interactuar con múltiples organizaciones, roles y servicios. Esta situación motiva la necesidad de proponer una solución que combine una arquitectura bien definida con prácticas modernas de desarrollo full-stack.

En este contexto, el presente artículo tiene como objetivo describir el diseño e implementación de una arquitectura MVC full-stack aplicada a un sistema de reservas multinegocio orientado a un modelo SaaS. La propuesta busca demostrar cómo la combinación de una arquitectura modular, el uso de patrones de diseño y tecnologías modernas de desarrollo web permite construir un sistema escalable, mantenible y preparado para su evolución futura.

2. MATERIALES Y METODO

A. Enfoque metodológico

El desarrollo del sistema se abordó mediante un enfoque de diseño modular, orientado a la separación de responsabilidades y a la mantenibilidad del software. Se aplicaron principios de la ingeniería de software y arquitectura de sistemas, priorizando la claridad estructural y la extensibilidad del sistema [12], [13]. La metodología adoptada se basa en la definición de una arquitectura clara desde las etapas iniciales, seguida de la implementación progresiva y validación funcional de los distintos componentes del sistema.

B. Arquitectura MVC full-stack

La arquitectura propuesta se fundamenta en el patrón Modelo–Vista–Controlador (MVC), aplicado a nivel de sistema completo. En este enfoque, la capa de modelo está representada por la base de datos relacional y la lógica de dominio, la capa de controlador corresponde al backend encargado de la gestión de reglas de negocio y seguridad, y la capa de vista está constituida por el frontend responsable de la interacción con el usuario. Este enfoque permite desacoplar las responsabilidades principales del sistema y facilita su evolución independiente [4], [5].

La implementación del backend y del frontend se apoyó en frameworks modernos ampliamente documentados, los cuales proporcionan soporte para arquitecturas modulares, manejo de dependencias y desarrollo full-stack [14], [15], [16].

C. Estructura del proyecto

El sistema Bokira se organiza bajo un enfoque monorepo, en el cual el backend y el frontend se mantienen como aplicaciones independientes dentro de una misma base de código. Esta decisión permite una gestión centralizada del proyecto, facilita la reutilización de componentes comunes y ofrece una visión integral del sistema. La organización del

proyecto refleja explícitamente la aplicación del patrón Modelo–Vista Controlador (MVC) a nivel de sistema completo, donde cada capa cumple una responsabilidad bien definida. En esta arquitectura, la capa de Modelo está representada por la base de datos y la lógica de dominio, la capa de Controlador corresponde al backend desarrollado con NestJS, encargado de gestionar los casos de uso, la seguridad y las reglas de negocio, y la capa de Vista está constituida por el frontend desarrollado con Next.js, responsable de la interacción con el usuario. Esta separación favorece el desacoplamiento entre capas y mejora la mantenibilidad del sistema [4], [5].

Se presenta una vista simplificada de la estructura general del proyecto Bokira. Esta organización evidencia la separación entre dominio, infraestructura y presentación, así como la correspondencia directa con las capas del patrón MVC aplicado a nivel full-stack.

```
bokira/
|-- apps/
|   |-- api/
|   |   |-- src/
|   |   |   |   common/
|   |   |   |-- modules/
|   |   |   |-- model/
|   |   |   `-- infra/
|   |   `-- prisma/schema.prisma
|   `-- web/
|       |-- app/
|       |-- components/
|       `-- lib/
`-- packages/shared/
```

Fig.1. Estructura del proyecto Bokira

La estructura presentada permite aislar la lógica de negocio de los detalles de persistencia y de la interfaz de usuario, facilitando la evolución independiente de cada capa. En particular, la separación entre las capas de dominio e infraestructura se alinea con los principios de arquitectura limpia, donde las dependencias se orientan hacia el núcleo del sistema, favoreciendo su mantenibilidad y extensibilidad [17].

D. Patrones de diseño aplicados

Con el objetivo de reforzar la modularidad del sistema y reducir el acoplamiento entre sus componentes, Bokira incorpora patrones de diseño ampliamente documentados en la literatura. Estos patrones se aplican tanto en el backend como en el frontend, permitiendo una arquitectura flexible y preparada para la evolución del sistema [10], [11].

1) *Patrón Repository:* El patrón Repository se emplea en el backend para abstraer el acceso a la base de datos y desacoplar la lógica de negocio de los detalles de persistencia. En Bokira, las interfaces de repositorio se definen en la capa de dominio (contratos), mientras que sus implementaciones concretas se ubican en la capa de infraestructura, permitiendo sustituir la tecnología de acceso a datos sin afectar el núcleo del sistema.

En la Fig.2. se muestra un ejemplo simplificado de una interfaz de repositorio para reservas, consumida por los casos de uso sin depender de Prisma u otra tecnología específica.

```
export interface BookingRepository {
  create(data: CreateBooking):
    Promise<Booking>;
  findById(id: string): Promise<Booking |
    null>;
}
```

Fig.2. Interfaz del repositorio de reservas (Repository)

Esta separación mejora la mantenibilidad y facilita pruebas, al permitir que los servicios de aplicación dependan de con tratos abstractos, coherente con prácticas de diseño orientadas al dominio [18].

2) *Patrón Facade:* En el frontend, el patrón Facade se utiliza para centralizar el acceso a servicios y evitar que los componentes de interfaz de usuario dependan directamente de detalles de red, rutas o serialización. Esta aproximación reduce la complejidad de los componentes visuales y permite evolucionar el acceso a datos sin impactar la capa de presentación.

La Fig.3. presenta una fachada que agrupa operaciones de consulta de servicios.

```
export const servicesApi = {
  getServices: () => fetchServices(),
  getServiceById: (id: string) =>
    fetchService(id)
};
```

Fig.3. Fachada de acceso a servicios (Facade)

3) *Patrón Observer:* El patrón Observer se aplica para manejar eventos del dominio y ejecutar acciones reactivas, como notificaciones asociadas a cambios de estado en una reserva. Este enfoque

desacopla la lógica principal (reservas) de efectos secundarios (notificaciones, auditoría o métricas), permitiendo añadir nuevos observadores sin modificar la lógica central.

La Fig.4. muestra un ejemplo simplificado de evento de dominio publicado cuando una reserva cambia a estado confirmado.

```
class BookingConfirmedEvent {
  constructor(public bookingId: string) {}
}
```

Fig.4. Evento de dominio para confirmación de reservas (Observer)

El uso de eventos y observadores favorece la extensibilidad del sistema y soporta la incorporación de nuevas funcionali dades sin introducir dependencias directas entre módulos [6].

3. RESULTADOS

La aplicación de la arquitectura MVC full-stack y de los patrones de diseño definidos en el presente trabajo dio como resultado la implementación de un sistema de reservas multinegocio con una estructura modular y coherente. Los resultados se presentan de manera progresiva, considerando primero el backend, luego el frontend y finalmente la integración general del sistema.

En el backend, se implementó una API organizada en módulos independientes, cada uno responsable de un con junto específico de funcionalidades, tales como la gestión de usuarios, negocios, servicios y reservas. Esta organización permitió incorporar mecanismos de autenticación, control de acceso y validación de reglas de negocio, manteniendo una separación clara entre la lógica de dominio y la infraestructura de persistencia. Los casos de uso se ejecutaron de forma independiente del motor de base de datos, a través de interfaces de repositorio previamente definidas.

En el frontend, se desarrolló una estructura basada en componentes reutilizables y en fachadas de acceso a servicios, lo que permitió integrar los flujos principales del sistema. Entre estos flujos se incluyeron el registro e inicio de sesión de usuarios, la visualización de negocios y servicios disponibles, y la creación y gestión de reservas. La comunicación con la API se realizó de manera consistente a través de una capa centralizada de acceso a datos.

La Tabla I resume las principales funcionalidades imple mentadas en el sistema como resultado del enfoque arquitectónico adoptado.

TABLA I. FUNCIONALIDADES IMPLEMENTADAS EN EL SISTEMA BOKIRA

Módulo	Funcionalidad
Usuarios	Registro, autenticación y perfil
Negocios	Gestión de información del negocio
Servicios	Creación y consulta de servicios
Reservas	Creación y gestión de reservas
Notificaciones	Eventos asociados a reservas

Finalmente, se implementó un mecanismo de eventos de dominio asociado a cambios de estado en las reservas, lo que permitió ejecutar acciones reactivas como base para notificaciones y extensiones futuras del sistema. Estos resultados evidenciaron la correcta integración entre las distintas capas del sistema bajo el enfoque MVC full-stack.

4. DISCUSIÓN

En relación con el objetivo planteado en la Introducción, orientado a describir y evaluar una arquitectura MVC full stack aplicada a un sistema de reservas multinegocio, los resultados obtenidos permitieron validar de manera cualitativa la hipótesis implícita de que una separación clara de responsabilidades contribuye a mejorar la organización y mantenibilidad del sistema.

Los resultados evidencian que la aplicación de una arquitectura MVC full-stack constituye un enfoque adecuado para sistemas de reservas que requieren soportar múltiples usuarios, organizaciones y flujos concurrentes. La modularidad observada en la estructura del sistema facilitó la incorporación progresiva de funcionalidades sin afectar componentes existen tes, en concordancia con principios ampliamente discutidos en la literatura sobre arquitectura de software [4], [5].

En comparación con arquitecturas monolíticas tradicionales, el enfoque adoptado en Bokira mostró ventajas en términos de separación de responsabilidades y claridad estructural. No

obstante, frente a arquitecturas basadas en microservicios, el modelo MVC full-stack presenta un menor nivel de complejidad operativa, lo que puede resultar adecuado para sistemas en etapas tempranas de evolución o con equipos de desarrollo reducidos [19].

Asimismo, al contrastar la propuesta con otros sistemas de gestión de reservas descritos en la literatura, se observa que la mayoría de las soluciones existentes priorizan la funcionalidad, pero no siempre integran de forma explícita principios arquitectónicos orientados a la mantenibilidad y extensibilidad del sistema [20]. En este sentido, la arquitectura propuesta busca equilibrar funcionalidad y calidad estructural.

Finalmente, si bien el sistema fue diseñado bajo un enfoque modular y extensible, el presente estudio se centró en una validación estructural y funcional. Futuros trabajos podrían incorporar evaluaciones cuantitativas de rendimiento, pruebas de carga y análisis de experiencia de usuario, con el fin de profundizar en la validación empírica del sistema en escenarios reales de uso.

5. CONCLUSION

El desarrollo del presente trabajo permitió demostrar que la aplicación de una arquitectura MVC full-stack constituye una alternativa adecuada para el diseño de sistemas de reservas multinegocio orientados a un modelo SaaS, al proporcionar una separación clara de responsabilidades entre las capas del sistema.

La organización modular de la arquitectura, junto con la aplicación de patrones de diseño, contribuyó a mejorar la mantenibilidad, comprensión y extensibilidad del sistema, aspectos clave en proyectos de software que requieren evolucionar de forma progresiva y controlada.

Desde la perspectiva del área de la ingeniería de software, la propuesta presentada refuerza la importancia de integrar principios arquitectónicos y patrones de diseño en el desarrollo de aplicaciones web modernas, particularmente en contextos multinegocio donde la complejidad estructural tiende a incrementarse.
Asimismo, los resultados obtenidos permiten afirmar que la arquitectura propuesta sienta una base sólida para la construcción de plataformas de reservas escalables y adaptables, sin introducir complejidad innecesaria en etapas tempranas del desarrollo.

Como trabajo futuro, se propone la realización de evaluaciones cuantitativas de rendimiento, pruebas de carga y estudios de experiencia de usuario, así como el análisis de enfoques arquitectónicos complementarios que permitan optimizar el sistema en escenarios de mayor escala y demanda.

6. RECONOCIMIENTOS

El autor agradece a la Universidad Nacional de Trujillo por el apoyo académico brindado durante el desarrollo del presente trabajo. Asimismo, expresa su reconocimiento al Ing. Jose Arturo Diaz Pulido por las orientaciones y recomendaciones proporcionadas a lo largo de la elaboración del artículo.

7. REFERENCIAS

[1] S. Law, «Online Reservation Systems: Design and Challenges,» *International Journal of Information Systems,* 2016.

[2] D. Parnas, «On the Criteria To Be Used in Decomposing Systems into Modules,» *Communications of the ACM,* 1972.

[3] M. Shaw y D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[4] L. Bass, P. Clements y R. Kazman, Software Architecture in Practice, Addison-Wesley, 2013.

[5] T. Reenskaug, «Models-Views-Controllers,» *IEEE Software,* 1979.

[6] F. Buschmann, R. Meunier, H. Rohnert y P. Sommerlad, Pattern-Oriented Software Architecture, Wiley, 1996.

[7] M. Armbrust y A. Fox, «A View of Cloud Computing,» *Communications of the ACM,* 2010.

[8] T. Erl, Cloud Computing: Concepts, Technology and Architecture, Prentice Hall, 2013.

[9] Q. Zhang y M. Chen, «Cloud Computing and SaaS Architecture,» *IEEE International Conference,* 2012.

[10] E. Gamma, R. Helm, R. Johnson y J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[11] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2002.

[12] R. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, 2014.

[13] I. Sommerville, Software Engineering, Pearson, 2016.

[14] «NestJS Documentation,» 2024. [En línea]. Available: https://docs.nestjs.com.

[15] «Next.js Documentation,» 2025. [En línea]. Available: https://nextjs.org/docs.

[16] «Prisma ORM Documentation,» 2025. [En línea]. Available: https://www.prisma.io/docs.

[17] R. Martin, Clean Architecture, Prentice Hall, 2017.

[18] E. Evans, Domain-Driven Design, Addison-Wesley, 2003.

[19] C. Richardson, Microservices Patterns, Manning, 2018.

[20] J. Pinedo, «Scheduling Systems and Applications,» *Springer,* 2012.

ANEXO:

I. DETALLES DE LA ESTRUCTURA E IMPLEMENTACIÓN DEL SISTEMA

El presente anexo incluye información técnica complementaria relacionada con la estructura del proyecto y algunos fragmentos representativos de la implementación del sistema Bokira. Estos elementos se presentan con el objetivo de ampliar detalles de diseño e implementación que, por razones de claridad y extensión, no se incluyeron en el cuerpo principal del artículo.

A. Estructura completa del proyecto

Se muestra una versión más detallada de la estructura del proyecto Bokira. Esta estructura evidencia la separación entre la lógica de dominio, la infraestructura y la capa de presentación, así como la organización modular del sistema bajo un enfoque MVC full-stack.

```
bokira/
|-- apps/
|   |-- api/
|   |   |-- src/
|   |   |   |-- common/          (guards,
    decorators, utilidades)
|   |   |   |-- modules/         (casos de
    uso)
|   |   |   |-- model/
|   |   |   |   |-- domain/      (entidades y
    reglas de negocio)
|   |   |   |   |-- ports/       (interfaces
    de repositorio)
|   |   |   |   `-- infra/
    (implementaciones)
|   |   |   `-- main.ts
|   |   `-- prisma/
|   |       `-- schema.prisma
|   `-- web/
|       |-- app/                 (pantallas y
    rutas)
|       |-- components/          (componentes
    de interfaz)
|       `-- lib/                 (fachadas y
    acceso a datos)
`-- packages/
    `-- shared/                  (DTOs y tipos
    compartidos)
```

Fig.5. Estructura detallada del proyecto Bokira

B. Ejemplo extendido del patrón Repository

Se presenta un ejemplo simplificado de la implementación concreta de un repositorio de reservas utilizando un ORM. Esta implementación cumple con el contrato definido en la capa de dominio y se ubica en la capa de infraestructura, permitiendo que la lógica de negocio permanezca independiente de los detalles de persistencia.

```
class PrismaBookingRepository implements
    BookingRepository {

  async create(data: CreateBooking):
    Promise<Booking> {
    return prisma.booking.create({ data });
  }

  async findById(id: string): Promise<Booking
    | null> {
    return prisma.booking.findUnique({
      where: { id }
    });
  }
}
```

Fig.6. Implementación del repositorio de reservas

C. *Ejemplo extendido del patrón Facade*

Se muestra una fachada utilizada en el frontend para centralizar el acceso a la API de servicios. Esta fachada abstrae los detalles de comunicación HTTP y permite que los componentes de interfaz interactúen con los servicios de manera uniforme.

```
export const servicesApi = {

  async getServices() {
    const response = await
    fetch('/api/services');
    return response.json();
  },

  async getServiceById(id: string) {
    const response = await
    fetch(`/api/services/${id}`);
    return response.json();
  }

};
```

Fig.7. Fachada de acceso a la API de servicios

D. *Ejemplo de evento de dominio*

Se presenta un ejemplo de evento de dominio asociado a la confirmación de una reserva. Este tipo de eventos permite desacoplar la lógica principal del sistema de acciones secundarias, como notificaciones o auditoría.

```
class BookingConfirmedEvent {
  constructor(public bookingId: string) {}
}
```

Fig.8. Evento de dominio para confirmación de reservas

II. CAPTURAS DE LA INTERFAZ WEB DEL SISTEMA

El presente anexo muestra capturas representativas de la interfaz web del sistema Bokira. Las imágenes incluidas tienen como finalidad ilustrar la capa de presentación del sistema y evidenciar su correspondencia con la arquitectura MVC full stack descrita en el cuerpo principal del artículo. Las capturas se presentan únicamente con fines ilustrativos y no contienen información sensible.

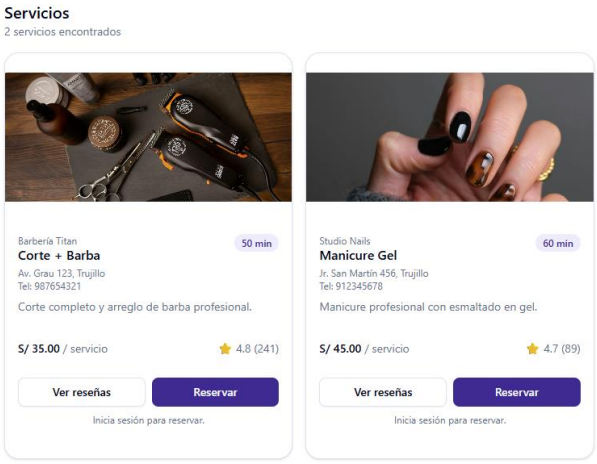

Fig.9. Vista principal del sistema Bokira mostrando el listado de servicios disponibles

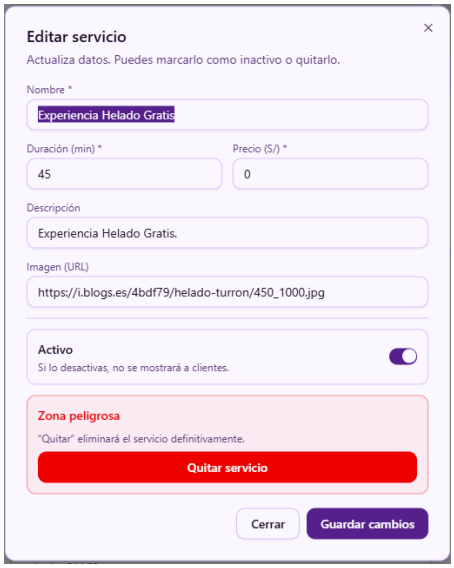

Fig.10. Interfaz de creación y gestión de reservas del sistema Bokira.

Fig.11. Vista administrativa para la gestión de servicios y reservas en Bokira.