Pontificia Universidad Javeriana
Departamento de Ingeniería de Sistemas
Arquitectura de Software

Presentación 1

Grupo 1

Luis Felipe Gutiérrez Rodríguez

Daniel Pérez Pinilla

María Paula Rodríguez Mancera

Profesor: Ing. Andrés Sánchez

Bogotá D.C, Octubre/2025

Tabla de Contenido

Análisis de Arquitectura Monolítica y Tecnologías Asociadas 4

1. Arquitectura Monolítica (Capas Monolito) 4

Definición 4

Características 5

Historia y Evolución 5

Ventajas y Desventajas 6

Casos de Uso 7

Casos de Aplicación 7

2. Jakarta EE + JSF (JavaServer Faces) 7

Definición 7

Características 8

Historia y Evolución 8

Ventajas y Desventajas 9

Casos de Uso 10

Casos de Aplicación 10

3. GlassFish y Payara 10

Definición 10

Características 11

Historia y Evolución 11

Ventajas y Desventajas 12

Casos de Uso 12

Casos de Aplicación 13

4. MariaDB 13

Definición 13

Características 13

Historia y Evolución 14

Ventajas y Desventajas 14

Casos de Uso 15

Casos de Aplicación 15
Relación entre los Temas Asignados 16
Qué tan común es el stack designado 16
Matriz de análisis de Principios SOLID vs Temas 17
Matriz de análisis de de Atributos de Calidad vs Temas 17
Matriz de análisis de Tácticas vs Temas 19
Matriz de análisis de Patrones vs Temas 20
Matriz de análisis de Mercado Laboral vs Temas 21
Ejemplo Práctico 21
Diagrama de Alto nivel 22
Modelo C4 24
Diagrama de Contexto 24
Diagrama de Contenedores 26
Diagrama de Componentes 26
Diagrama de Código 28
Diagrama Dinámico C4 29
Diagrama de Despliegue C4 31
Diagrama de Paquetes UML 33
Conclusiones 34
Referencias: 35

Tabla de Figuras

Imagen 1. Diagrama de Alto nivel de las Reservas de Hotel 23
Imagen 2. Diagrama de Contexto de las reservas de Hotel. 25
Imagen 3. Diagrama de Contenedores de las reservas de Hotel. 26
Imagen 4. Diagrama de Componentes de las reservas de Hotel. 27
Imagen 5. Diagrama de Código de las reservas de Hotel. 28
Imagen 6. Diagrama Dinámico de las reservas de Hotel. 30
Imagen 7. Diagrama de Despliegue de las reservas de Hotel. 32
Imagen 8. Diagrama de Paquetes UML de las reservas de Hotel 33

Análisis de Arquitectura Monolítica y Tecnologías Asociadas

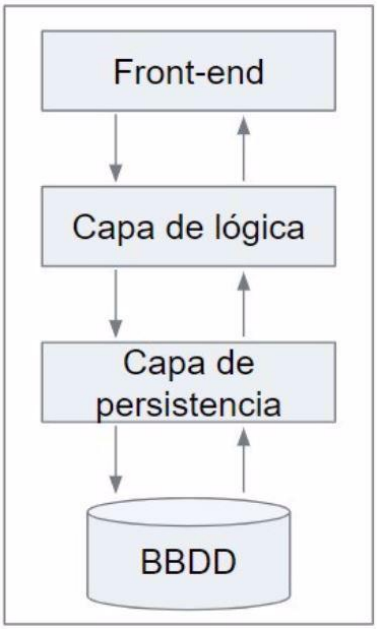
En esta sección se presenta un análisis exhaustivo de las tecnologías clave asociadas a la arquitectura monolítica, un enfoque ampliamente utilizado para la construcción de aplicaciones empresariales. Se ha investigado y detallado cómo la arquitectura monolítica, especialmente en su variante de Capas Monolito, se implementa en aplicaciones de gran escala, destacando sus ventajas, desventajas y los casos de uso más comunes. Además, se ha explorado el papel fundamental que juegan tecnologías como Jakarta EE (anteriormente Java EE) y su componente JSF (JavaServer Faces) en la construcción de aplicaciones modulares dentro de una arquitectura monolítica.

La investigación también abarca dos componentes cruciales en el despliegue de estas aplicaciones: GlassFish y Payara, que son servidores de aplicaciones que implementan Jakarta EE y permiten ejecutar aplicaciones empresariales de alta demanda. Se discuten sus características, historia, y la evolución de cada uno, destacando la relevancia de Payara como una versión mejorada de GlassFish.

Por último, se presenta un análisis sobre MariaDB, una base de datos relacional que se utiliza en aplicaciones monolíticas para gestionar la persistencia de datos, explorando sus características, ventajas y su importancia en el contexto de sistemas empresariales y aplicaciones web.

1. Arquitectura Monolítica (Capas Monolito)

Definición



La arquitectura monolítica es un enfoque de diseño de software en el cual toda la funcionalidad de una aplicación (desde la interfaz de usuario hasta la lógica de negocio y la persistencia de datos) está integrada en un solo bloque o unidad. Esto significa que todas las partes de la aplicación están fuertemente acopladas y compartiendo el mismo espacio de memoria. En una arquitectura monolítica tradicional, no existe separación entre componentes o servicios independientes, lo que puede hacer que el desarrollo y el despliegue sean más rápidos y sencillos en las primeras fases del proyecto.

La arquitectura Capas Monolito, una variante de la monolítica, organiza esta unidad en capas. Este patrón organiza las funcionalidades de la aplicación en capas distintas, aunque todas siguen siendo parte de una misma unidad desplegada. Las capas típicas en una arquitectura Capas Monolito son:

- **Capa de presentación** (Interfaz de usuario): Maneja la interacción con el usuario, mostrando la información y recibiendo entradas.
- **Capa de lógica de negocio**: Contiene la lógica que controla el flujo de la aplicación, realiza el procesamiento de datos y gestiona las reglas de negocio.
- **Capa de acceso a datos**: Se encarga de la interacción con la base de datos y otros sistemas de almacenamiento, manejando la persistencia de datos.

A pesar de la separación en capas, todo el sistema se encuentra dentro de una misma aplicación o proceso, lo que hace que no haya una verdadera separación entre estos componentes.

Características

1. **Despliegue único**: En una arquitectura monolítica, toda la aplicación se despliega como una unidad. Esto simplifica el proceso de implementación y administración, ya que no se necesita manejar múltiples servicios o instancias. Sin embargo, esto también implica que, si alguna parte del sistema necesita ser actualizada, toda la aplicación debe ser desplegada nuevamente, lo que puede generar tiempos de inactividad.
2. **Cohesión**: En una arquitectura monolítica, las funcionalidades y componentes de la aplicación están **altamente acoplados**. Esto significa que los módulos o capas dependen entre sí, lo que facilita la comunicación interna dentro de la aplicación, pero dificulta la modificación de una parte del sistema sin afectar el resto de la aplicación. Las aplicaciones monolíticas suelen compartir el mismo espacio de memoria y recursos, lo que las hace cohesivas, pero también propensas a problemas cuando se intenta realizar cambios importantes.
3. **Escalabilidad limitada**: Escalar una aplicación monolítica implica duplicar toda la aplicación en vez de solo los componentes que requieren mayor capacidad. Por ejemplo, si solo la capa de procesamiento de datos necesita más recursos, el modelo monolítico obliga a duplicar toda la aplicación, incluyendo la capa de presentación y la de lógica de negocio, lo cual no es eficiente y limita la capacidad de escalar por partes.
4. **Mantenimiento complejo a largo plazo**: A medida que la aplicación crece, el código monolítico se vuelve más difícil de gestionar. La adición de nuevas funcionalidades o la modificación de características existentes puede provocar efectos secundarios en otras partes del sistema. Esto puede generar dificultades en la prueba y en la detección de errores, ya que todo el sistema está interrelacionado. Además, las dependencias entre las capas pueden hacer que los cambios sean costosos y arriesgados.

Historia y Evolución

La arquitectura monolítica ha sido el patrón de diseño dominante desde los primeros días de la informática, principalmente porque era la forma más sencilla y eficiente de construir aplicaciones en entornos con hardware limitado. A partir de la década de 1960, la mayoría de las aplicaciones

eran monolíticas debido a la falta de infraestructura tecnológica avanzada y la ausencia de metodologías para dividir aplicaciones en componentes modulares.

Sin embargo, con el tiempo, las aplicaciones fueron creciendo en tamaño y complejidad, lo que llevó a la necesidad de arquitecturas más escalables y flexibles. En la década de 2000, el patrón de **microservicios** surgió como una alternativa para abordar las limitaciones de la arquitectura monolítica, especialmente en lo que respecta a la escalabilidad y el mantenimiento. Mientras tanto, la arquitectura monolítica sigue siendo relevante y útil en ciertos contextos, como aplicaciones de pequeña y mediana escala, o en entornos de software heredado donde la migración a microservicios no es viable.

Ventajas y Desventajas

Ventajas:

1. **Desarrollo inicial rápido**: Permite un desarrollo más rápido inicialmente, dado que no hay que preocuparse por la comunicación entre múltiples servicios. Todo está contenido en una única base de código, lo que facilita la creación rápida de un producto funcional.
2. **Simplicidad**: Es menos compleja en términos de diseño y despliegue, lo que la hace adecuada para proyectos pequeños o con recursos limitados. La integración de componentes no requiere la implementación de servicios de comunicación entre aplicaciones ni protocolos complicados.
3. **Facilidad en pruebas**: Al ser una sola unidad, las pruebas unitarias e integradas son más fáciles de implementar, ya que no hay dependencias externas o servicios separados que deban probarse de manera independiente.

Desventajas:

1. **Escalabilidad limitada**: Como se mencionó, la escalabilidad en una arquitectura monolítica es compleja. En lugar de escalar partes específicas del sistema (como la base de datos o el frontend), la aplicación completa debe ser replicada, lo que es ineficiente.
2. **Dificultad para gestionar grandes sistemas**: Con el crecimiento de la aplicación, el código monolítico puede volverse muy difícil de manejar. La interacción entre las diferentes partes del sistema puede ser compleja, y el código de la aplicación puede volverse acoplado y difícil de cambiar sin afectar otras partes.
3. **Despliegue y mantenimientos complicados**: Al tener que actualizar todo el sistema de una vez, los ciclos de despliegue y mantenimiento pueden volverse largos y arriesgados. Esto implica que un solo error puede afectar a toda la aplicación.

Casos de Uso

La arquitectura monolítica es ideal en los siguientes casos:

1. **Aplicaciones pequeñas o medianas**: En proyectos pequeños o medianos, donde la aplicación no tiene una enorme carga de trabajo o necesidad de escalabilidad, una arquitectura monolítica puede ser la opción más adecuada.
2. **Sistemas heredados**: Muchas aplicaciones que fueron creadas hace años siguen siendo monolíticas, y migrarlas a una arquitectura de microservicios no siempre es viable. Estos sistemas suelen ser de gran tamaño y complejidad, pero siguen funcionando bien para las necesidades de la empresa.
3. **Proyectos de inicio rápido**: En startups o proyectos donde el objetivo es lanzar rápidamente un producto funcional, la arquitectura monolítica es una opción sencilla y directa para evitar la complejidad inicial de una arquitectura distribuida.

Casos de Aplicación

1. **Sistemas ERP tradicionales**: Empresas que usan aplicaciones monolíticas para gestionar su contabilidad, inventarios y otros procesos empresariales. Ejemplos incluyen sistemas como **SAP ERP** en su versión monolítica, donde todos los módulos (finanzas, inventario, ventas, etc.) están integrados en una sola aplicación.
2. **Aplicaciones bancarias**: Muchas instituciones financieras han utilizado durante años aplicaciones monolíticas para gestionar transacciones internas. Estos sistemas se caracterizan por tener componentes como bases de datos centralizadas y procesos de pagos que son interdependientes dentro de una sola unidad.

2. Jakarta EE + JSF (JavaServer Faces)

Definición

Jakarta EE (anteriormente conocido como Java EE) es una plataforma de desarrollo para aplicaciones empresariales basada en Java. Consiste en un conjunto de especificaciones que definen cómo los componentes y servicios deben interactuar dentro de una aplicación empresarial. Jakarta EE proporciona un marco de trabajo robusto y flexible para crear aplicaciones distribuidas, escalables, y seguras, que pueden ejecutarse en una variedad de servidores de aplicaciones. Las especificaciones de Jakarta EE cubren una amplia gama de

aspectos, como la persistencia de datos (a través de JPA), la gestión de transacciones, la seguridad y la mensajería, entre otros.

JSF (JavaServer Faces) es una tecnología dentro de Jakarta EE que facilita la creación de interfaces de usuario web para aplicaciones empresariales. JSF ofrece una estructura basada en componentes, lo que significa que permite el desarrollo de interfaces de usuario mediante componentes visuales reutilizables. Esto ayuda a simplificar la creación de aplicaciones web al integrar la presentación con la lógica de negocio de forma eficiente y modular.

Características

1. **Modularidad**: Jakarta EE permite crear aplicaciones modulares mediante el uso de diversos componentes, como EJB (Enterprise JavaBeans), JPA (Java Persistence API), JMS (Java Message Service) y JSF. Cada uno de estos componentes está diseñado para abordar diferentes aspectos de la arquitectura de software. La modularidad permite que los desarrolladores construyan aplicaciones de forma más organizada y reutilizable.
2. **Integración**: Jakarta EE facilita la integración con otras tecnologías y servicios. Por ejemplo, JPA permite la persistencia de datos en bases de datos relacionales, JMS facilita la mensajería asincrónica, y EJB ofrece la lógica de negocio centralizada. Estas tecnologías se pueden usar de manera conjunta en una aplicación, brindando flexibilidad y escalabilidad.
3. **Desarrollo basado en componentes**: JSF utiliza un modelo de componentes visuales para construir la interfaz de usuario. Cada componente de JSF está vinculado a la lógica de negocio que se ejecuta en el backend, lo que permite una integración más fluida entre la interfaz de usuario y el servidor. Además, los componentes son reutilizables y pueden personalizarse fácilmente, lo que mejora la productividad en el desarrollo.
4. **Compatibilidad con servidores de aplicaciones**: Jakarta EE es completamente compatible con servidores de aplicaciones como GlassFish y Payara. Estos servidores implementan las especificaciones de Jakarta EE y proporcionan un entorno adecuado para ejecutar aplicaciones empresariales escalables y seguras. Estos servidores también proporcionan servicios adicionales como gestión de transacciones, seguridad y escalabilidad, lo que hace que Jakarta EE sea adecuado para aplicaciones de misión crítica.

Historia y Evolución

Java EE, desarrollado originalmente por Sun Microsystems, fue lanzado en 1999 con el objetivo de estandarizar el desarrollo de aplicaciones empresariales basadas en Java. Java EE ofreció una serie de especificaciones que ayudaron a simplificar el desarrollo de aplicaciones grandes y complejas, al mismo tiempo que proporcionaban un entorno escalable y seguro. A lo largo de los

años, Java EE fue evolucionando, y con el tiempo se convirtió en una de las plataformas más populares para la creación de aplicaciones empresariales.

En 2017, Oracle, el propietario de Java EE, transfirió el proyecto a la Eclipse Foundation, una organización sin fines de lucro que gestiona proyectos de código abierto. El proyecto pasó a llamarse Jakarta EE debido a un conflicto con Oracle sobre los derechos de nombre. Esta transición marcó un nuevo capítulo para Java EE, permitiendo una mayor participación de la comunidad y una evolución más abierta. Como parte de esta transición, JSF continuó siendo una tecnología clave dentro de Jakarta EE, facilitando el desarrollo de aplicaciones web interactivas.

Ventajas y Desventajas

Ventajas:

1. **Estándares abiertos**: Jakarta EE se basa en estándares abiertos, lo que garantiza su interoperabilidad con otras plataformas y sistemas. Al ser una especificación estándar, los desarrolladores pueden estar seguros de que las aplicaciones construidas con Jakarta EE pueden ser ejecutadas en cualquier servidor de aplicaciones compatible.
2. **Productividad del desarrollador**: JSF proporciona una serie de componentes reutilizables para la construcción de interfaces de usuario, lo que mejora la productividad al evitar la necesidad de crear desde cero componentes comunes como formularios, tablas y botones. Además, su integración con JPA y EJB permite una construcción más rápida de aplicaciones completas.
3. **Escalabilidad**: Jakarta EE está diseñado para manejar aplicaciones empresariales de gran escala, proporcionando herramientas y servicios para gestionar la carga, las transacciones y los recursos del sistema de manera eficiente. Esta arquitectura permite a las aplicaciones escalar conforme crecen las necesidades de la empresa.

Desventajas:

1. **Curva de aprendizaje**: Para los desarrolladores que no están familiarizados con las tecnologías de Java EE o Jakarta EE, la curva de aprendizaje puede ser empinada. La plataforma es extensa y tiene muchas especificaciones y herramientas que deben aprenderse. Además, la integración de múltiples tecnologías (como JPA, EJB y JMS) puede resultar compleja al principio.
2. **Rendimiento**: Aunque Jakarta EE es eficiente, su carga de trabajo puede ser más alta que la de otras plataformas más ligeras, especialmente cuando se usa en aplicaciones pequeñas. La naturaleza robusta de Jakarta EE y su enfoque modular pueden generar sobrecarga, lo que puede afectar el rendimiento en aplicaciones con requisitos estrictos de latencia.

Casos de Uso

1. **Aplicaciones empresariales**: Jakarta EE es ideal para el desarrollo de aplicaciones empresariales grandes y complejas que requieren un alto nivel de seguridad, escalabilidad y gestión de transacciones. Ejemplos incluyen sistemas de gestión empresarial, aplicaciones de comercio electrónico y plataformas de servicios financieros.
2. **Sistemas con múltiples capas**: Jakarta EE es perfecto para aplicaciones que tienen una arquitectura multicapa. Estas aplicaciones separan la lógica de presentación, la lógica de negocio y el acceso a datos, lo que permite un desarrollo más organizado y escalable. Ejemplos incluyen sistemas de ERP (Enterprise Resource Planning) y CRM (Customer Relationship Management).

Casos de Aplicación

1. **Airbus**: Airbus utiliza Jakarta EE para gestionar sistemas internos de gestión de proyectos y operaciones. La capacidad de Jakarta EE para manejar aplicaciones empresariales complejas y de gran escala es esencial para mantener las operaciones de una empresa tan grande y diversa.
2. **Payara y GlassFish**: Payara Server y GlassFish son dos ejemplos de servidores de aplicaciones que implementan Jakarta EE. Ambos se utilizan en aplicaciones empresariales de gran escala en sectores como la banca, la salud y las telecomunicaciones, donde la fiabilidad y la capacidad de escalar son fundamentales. Estos servidores proporcionan características avanzadas como gestión de transacciones distribuidas, seguridad y escalabilidad, que son esenciales para las aplicaciones críticas.

3. GlassFish y Payara

Definición

GlassFish es un servidor de aplicaciones de código abierto que implementa la plataforma Jakarta EE (anteriormente conocida como Java EE). GlassFish se utiliza para desplegar aplicaciones empresariales Java y proporciona todas las especificaciones necesarias para ejecutar aplicaciones que aborden necesidades como persistencia de datos, gestión de transacciones, mensajería y seguridad.

Payara, por otro lado, es un fork (una bifurcación) de GlassFish. Fue creado por un grupo de ex-desarrolladores de GlassFish con el objetivo de ofrecer una versión más robusta, segura y con soporte

extendido de GlassFish. Aunque Payara conserva la compatibilidad total con Jakarta EE, ha sido optimizado para mejorar el rendimiento, proporcionar actualizaciones periódicas y ofrecer un mejor soporte empresarial.

Ambos servidores son ampliamente utilizados para implementar aplicaciones empresariales que requieren un entorno robusto y seguro, especialmente cuando las aplicaciones se desarrollan sobre Jakarta EE.

Características

1. **Soporte completo de Jakarta EE**: Implementan completamente la plataforma Jakarta EE. Esto significa que ambos servidores son capaces de soportar todas las especificaciones estándar de Jakarta EE, como JPA (Java Persistence API), JSF (JavaServer Faces), EJB (Enterprise JavaBeans), JMS (Java Message Service), entre otras. Esto los hace adecuados para aplicaciones empresariales de gran envergadura y de misión crítica.
2. **Escalabilidad**: GlassFish y Payara están diseñados para ser escalables, lo que significa que pueden manejar aplicaciones de gran escala. Esto es fundamental para empresas que necesitan una infraestructura capaz de escalar con el crecimiento de la aplicación, ya sea aumentando la cantidad de usuarios, transacciones o datos procesados. Estos servidores ofrecen gestión de clústeres, que permite distribuir la carga de trabajo entre varias instancias del servidor.
3. **Comunidad activa**: Payara cuenta con una comunidad activa de desarrolladores, usuarios y profesionales que contribuyen al proyecto. Esta comunidad brinda soporte extendido y se encarga de mantener y actualizar el servidor de manera constante. Además, Payara ofrece un servicio de soporte empresarial para clientes que necesitan una atención más especializada, lo que lo convierte en una opción preferida por empresas que requieren actualizaciones frecuentes y soporte técnico confiable.

Historia y Evolución

GlassFish fue desarrollado originalmente por Sun Microsystems y lanzado en 2005 como un servidor de aplicaciones de código abierto para Java EE. Fue diseñado como un servidor de referencia para implementar las especificaciones de Java EE y servir como base para otras implementaciones de servidores de aplicaciones Java. En 2010, Oracle adquirió Sun Microsystems y continuó el desarrollo de GlassFish.

En 2015, el proyecto GlassFish pasó a ser gestionado por la comunidad de Payara. El equipo de desarrollo de Payara lanzó Payara Server como una bifurcación de GlassFish con el objetivo de mejorar el rendimiento, añadir características de seguridad, y ofrecer soporte empresarial a largo plazo. Desde entonces, Payara Server se ha mantenido como una alternativa más robusta a GlassFish, con actualizaciones y parches constantes.

Ventajas y Desventajas

Ventajas:

1. **Código abierto**: Ambos servidores, GlassFish y Payara, son gratuitos y de código abierto, lo que permite a las empresas utilizarlos sin necesidad de pagar licencias caras. Además, los desarrolladores tienen la capacidad de modificar el código según sus necesidades específicas, lo que proporciona gran flexibilidad.
2. **Soporte empresarial**: Payara ofrece soporte empresarial con actualizaciones regulares, parches de seguridad y asistencia técnica. Este servicio es ideal para organizaciones que necesitan garantizar la estabilidad y seguridad a largo plazo de sus aplicaciones, especialmente en sectores como el bancario o telecomunicaciones.
3. **Actualizaciones constantes**: A diferencia de GlassFish, que no recibe actualizaciones regulares por parte de Oracle, Payara se actualiza continuamente, con parches de seguridad y nuevas características, lo que lo hace más adecuado para empresas que necesitan un servidor confiable y con soporte activo.

Desventajas:

1. **Complejidad**: Ambos servidores requieren una configuración adecuada y mantenimiento constante, especialmente en aplicaciones de gran escala. La gestión de clústeres, el balanceo de carga y la configuración de seguridad pueden resultar complejos, especialmente para desarrolladores novatos o equipos con poca experiencia en servidores de aplicaciones.
2. **Sobrecarga**: Dado que Payara y GlassFish son servidores completos que implementan todas las especificaciones de Jakarta EE, su sobrecarga puede ser significativa, especialmente en aplicaciones pequeñas o con recursos limitados. Para aplicaciones más ligeras o con requisitos de rendimiento extremadamente altos, estos servidores podrían no ser la opción más eficiente.

Casos de Uso

1. **Sistemas empresariales grandes**: GlassFish y Payara son ideales para aplicaciones empresariales de gran escala que requieren alta disponibilidad, confiabilidad y seguridad. Esto incluye sistemas de gestión de recursos empresariales (ERP), plataformas de banca electrónica o sistemas de comercio electrónico que requieren procesamiento transaccional a gran escala.
2. **Aplicaciones de misión crítica**: Empresas que gestionan aplicaciones de misión crítica—como bancos, empresas de telecomunicaciones, o proveedores de servicios públicos—necesitan servidores que ofrezcan alta disponibilidad y seguridad. Payara y GlassFish están

diseñados para soportar aplicaciones que no pueden permitirse tiempos de inactividad o fallos en sus servicios.

Casos de Aplicación

1. **Telefónica**: U a de las mayores compañías de telecomunicaciones del mundo, utiliza Payara para gestionar sus servicios empresariales internos. La capacidad de Payara para manejar aplicaciones a gran escala y su soporte extendido le permiten a Telefónica operar servicios críticos con alta disponibilidad y seguridad.
2. **Banco Santander**: Utriliza GlassFish en la implementación de algunas de sus aplicaciones internas. La implementación de Jakarta EE a través de GlassFish permite que el banco gestione procesos bancarios complejos con eficiencia, asegurando una correcta integración de su sistema de servicios y la gestión de transacciones de manera segura y escalable.

4. MariaDB

Definición

MariaDB es un sistema de gestión de bases de datos relacional de código abierto, que se utiliza para almacenar y gestionar datos en aplicaciones web y empresariales. MariaDB es totalmente compatible con MySQL, lo que significa que las aplicaciones que fueron diseñadas para MySQL pueden migrar fácilmente a MariaDB sin necesidad de modificar su código. Además de ser compatible con MySQL, MariaDB ha introducido algunas mejoras de rendimiento y características adicionales que lo hacen adecuado para aplicaciones a gran escala que requieren alta disponibilidad y un alto rendimiento en operaciones de lectura y escritura.

Características

1. **Compatible con MySQL**: Mantiene compatibilidad total con MySQL, lo que permite a los desarrolladores y administradores de bases de datos cambiar de MySQL a MariaDB sin necesidad de realizar modificaciones en el código de la aplicación. Esta compatibilidad incluye el mismo conjunto de comandos SQL, interfaces de programación de aplicaciones (APIs) y estructuras de almacenamiento.
2. **Rendimiento optimizado**: Se ha optimizado para proporcionar un rendimiento superior en operaciones de lectura y escritura. A lo largo de los años, ha mejorado las capacidades de almacenamiento, índices y ejecución de consultas. Por ejemplo, MariaDB utiliza el

motor de almacenamiento Aria, que mejora el rendimiento de las operaciones en bases de datos con grandes volúmenes de datos.

3. **Escalabilidad**: Es adecuado tanto para aplicaciones pequeñas como para grandes sistemas empresariales que requieren bases de datos escalables. MariaDB ofrece características avanzadas como replicación y clustering que permiten gestionar grandes volúmenes de datos distribuidos y mejorar la disponibilidad y fiabilidad de las aplicaciones.
4. **Alta disponibilidad**: Soporta diversas técnicas de alta disponibilidad, como la replicación maestro-esclavo, la replicación multimaster, y el clustering Galera, lo que permite que las bases de datos estén siempre disponibles, incluso en situaciones de fallos de hardware o problemas en los servidores. Esto es fundamental para aplicaciones críticas donde el tiempo de inactividad debe minimizarse.

Historia y Evolución

MariaDB fue creada por los desarrolladores originales de MySQL, liderados por Michael "Monty" Widenius, después de que Oracle Corporation adquiriera MySQL AB en 2009. Ante la incertidumbre sobre el futuro de MySQL bajo Oracle, Widenius y su equipo decidieron crear MariaDB como una alternativa de código abierto y libre a MySQL, asegurando que la comunidad tuviera un control total sobre el desarrollo del sistema de bases de datos.

Desde su creación, MariaDB ha ganado popularidad como un fork de MySQL y se ha convertido en una de las opciones preferidas para la gestión de bases de datos de código abierto debido a sus mejoras de rendimiento y seguridad. MariaDB está siendo ampliamente adoptada por empresas, como Wikipedia y Google, que buscan soluciones de bases de datos robustas y de alto rendimiento.

Ventajas y Desventajas

Ventajas:

1. **Código abierto**: Es un sistema de código abierto, lo que significa que es totalmente gratuito para su uso, modificación y distribución. Además, la comunidad activa de MariaDB contribuye constantemente al proyecto, mejorando su rendimiento, seguridad y características. Esto lo convierte en una opción ideal para empresas que buscan soluciones de bases de datos sin los costos asociados a las licencias de software propietario.
2. **Alta disponibilidad**: Orece características avanzadas para garantizar la alta disponibilidad de los datos. La replicación y el clustering permiten que las aplicaciones sigan funcionando sin interrupciones, incluso en situaciones de fallo del servidor, lo cual es crucial para aplicaciones que no pueden permitirse tiempo de inactividad.
3. **Rendimiento mejorado**: Gracias a mejoras en los motores de almacenamiento y optimizaciones internas, MariaDB ofrece un rendimiento superior en operaciones de lectura y escritura en comparación con MySQL. Esto es especialmente valioso en

aplicaciones que manejan grandes volúmenes de datos o tienen necesidades de alta carga transaccional.

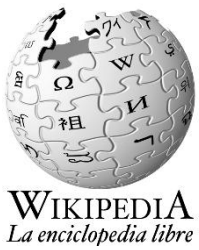
Desventajas:

1. **Requiere gestión**: Aunque MariaDB es un sistema de bases de datos robusto y de alto rendimiento, su administración puede ser más compleja en grandes aplicaciones. La configuración y mantenimiento de clusters, replicación y optimización de consultas puede requerir una cantidad significativa de conocimientos técnicos. Para aplicaciones más pequeñas o con menos experiencia en administración de bases de datos, esto podría ser una barrera.
2. **Compatibilidad limitada con algunas características de MySQL**: Aunque MariaDB es totalmente compatible con MySQL, algunas implementaciones específicas de MySQL no siempre funcionan de la misma manera en MariaDB debido a las mejoras y cambios que se han introducido en el sistema. Esto puede generar problemas de compatibilidad cuando se migra de MySQL a MariaDB.

Casos de Uso

1. **Aplicaciones web y móviles**: Es ideal para aplicaciones web y móviles que requieren bases de datos relacionales. Su capacidad para manejar grandes volúmenes de datos y ofrecer un alto rendimiento lo hace adecuado para sitios web de alto tráfico, plataformas de comercio electrónico y aplicaciones móviles que necesitan gestionar grandes cantidades de usuarios y datos.
2. **Sistemas empresariales**: Es ampliamente utilizado en sistemas empresariales que requieren bases de datos rápidas y eficientes. Desde aplicaciones de gestión de recursos empresariales (ERP) hasta plataformas de gestión de relaciones con clientes (CRM), MariaDB es adecuado para cualquier sistema que necesite realizar operaciones transaccionales complejas con alta disponibilidad y rendimiento óptimo.

Casos de Aplicación



1. **Wikipedia**: Una de las bases de datos más grandes del mundo, utiliza MariaDB para gestionar su vasta colección de artículos y metadatos. MariaDB se ha convertido en una opción ideal para Wikipedia debido a su rendimiento optimizado en consultas de lectura y su capacidad para manejar grandes volúmenes de datos de manera eficiente.
2. **Google**: Utiliza MariaDB en ciertos servicios internos. Aunque Google emplea una combinación de tecnologías para gestionar

sus grandes volúmenes de datos, MariaDB ha sido adoptada en algunos de sus sistemas debido a su fiabilidad, escalabilidad y compatibilidad con MySQL.

Relación entre los Temas Asignados

En esta sección se abordará la relación entre los temas asignados en el contexto de la arquitectura monolítica, Jakarta EE, JSF, GlassFish, Payara, y MariaDB. A través de una serie de análisis, se evaluará la comunidad del stack designado en el desarrollo de aplicaciones empresariales, y cómo las tecnologías relacionadas se alinean con los principios y patrones de diseño de software. Para facilitar esta comprensión, se presentarán diversas matrices de análisis, cada una de las cuales examina la interacción de los temas desde distintas perspectivas, tales como la compatibilidad con los principios SOLID, los atributos de calidad del software, las tácticas de diseño, los patrones de software y el mercado laboral.

Cada una de estas matrices servirá para proporcionar una visión detallada de cómo estas tecnologías se interrelacionan y contribuyen al desarrollo de soluciones robustas y escalables. A continuación, se presentan los distintos enfoques de análisis para ilustrar la relación y relevancia de los temas en el diseño de aplicaciones empresariales.

Qué tan común es el stack designado

Este stack designado sigue siendo una opción ampliamente adoptada en muchas empresas, especialmente aquellas que priorizan la estabilidad y buscan un conjunto de tecnologías maduras y probadas. Aunque en los últimos años han ganado terreno arquitecturas basadas en microservicios, el stack monolítico sigue siendo preferido en empresas que operan en sectores más tradicionales, como la banca, la telecomunicación u organizaciones grandes que ya tienen infraestructuras establecidas. La arquitectura monolítica proporciona un camino directo hacia el desarrollo de aplicaciones completas, con menos complejidad inicial en términos de configuración de servicios y despliegue, lo que lo convierte en una opción atractiva para proyectos pequeños a medianos o para sistemas heredados que aún requieren mantenimiento y evolución.

Además, tecnologías como Jakarta EE y JSF tienen un largo historial de implementación en aplicaciones empresariales, con un amplio soporte y comunidad, lo que otorga a los desarrolladores una base sólida para crear soluciones escalables y de alta calidad. Payara y GlassFish ofrecen un soporte extendido y fiabilidad adicional, especialmente en aplicaciones que requieren alta disponibilidad y transacciones seguras. MariaDB, por su parte, es una base de datos confiable y eficiente, con una compatibilidad total con MySQL, lo que facilita la transición entre ambas tecnologías y permite a las empresas manejar grandes volúmenes de datos sin comprometer el rendimiento.

Matriz de análisis de Principios SOLID vs Temas

Los principios SOLID son un conjunto de principios de diseño orientados a objetos que promueven la mantenibilidad y flexibilidad del código. Estos principios son fundamentales para la creación de aplicaciones robustas y de alta calidad, y se basan en la idea de que el software debe ser fácil de modificar, extender y mantener con el tiempo. Además, son conocidos por facilitar la creación de sistemas más modulares y desacoplados. A continuación, se presenta una matriz que analiza cómo los temas asignados se alinean con estos principios:

Principio SOLID	Arquitectura Monolítica (Capas Monolito)	Jakarta EE + JSF	GlassFish/Payara	MariaDB
S: *Single Responsibility*	Cumple con la separación de responsabilidades entre capas.	JSF permite separar la lógica de presentación del backend.	Permite la separación de la lógica de servidor y cliente.	Gestiona exclusivamente la persistencia de datos.
O: *Open/Closed*	El monolito puede ser difícil de extender sin modificar el código existente.	Jakarta EE permite extensión mediante EJB, JPA, etc.	Payara facilita la adición de módulos sin modificar el núcleo.	Fácil extensión con nuevas tablas y procedimientos.
L: *Liskov Substitution*	Difícil de implementar debido al alto acoplamiento entre capas.	JSF facilita la implementación de componentes intercambiables.	Soporta implementación de servicios adicionales.	No aplica directamente, ya que es un sistema de bases de datos.
I: *Interface Segregation*	Puede tener interfaces complejas debido al enfoque monolítico.	Jakarta EE permite la separación de interfaces y servicios.	Permite integrar distintas interfaces a través de módulos.	No aplicable directamente, es un sistema de gestión de bases de datos.
D: *Dependency Inversion*	La dependencia entre las capas puede ser difícil de manejar.	Jakarta EE facilita la inyección de dependencias a través de CDI.	GlassFish y Payara soportan la inyección de dependencias.	MariaDB gestiona dependencias de datos, pero no tiene un equivalente directo.

Matriz de análisis de de Atributos de Calidad vs Temas

Los atributos de calidad son características que determinan el rendimiento y la efectividad de una aplicación. Algunos de los atributos más importantes incluyen la escabilidad, el rendimiento, la

seguridad, la mantenibilidad y la disponibilidad. Estos atributos son esenciales para garantizar que el software no solo funcione correctamente, sino que también sea capaz de soportar cargas de trabajo más altas, resistir fallos y adaptarse a cambios en los requisitos del sistema. A continuación, se presenta una matriz que analiza cómo las tecnologías asignadas abordan estos atributos:

Atributo de Calidad	Arquitectura Monolítica (Capas Monolito)	Jakarta EE + JSF	GlassFish / Payara	MariaDB
Compatibilidad	Limitada, ya que el sistema suele estar fuertemente acoplado. Dificulta la integración con otros servicios o APIs externas.	Alta compatibilidad gracias a las especificaciones estándar de Jakarta EE. Permite interoperar con otros servicios y frameworks Java.	Totalmente compatibles con Jakarta EE, facilitando la interoperabilidad entre módulos y aplicaciones empresariales.	Compatible con MySQL y múltiples lenguajes de programación, lo que permite integrarse fácilmente en diferentes entornos.
Usabilidad	Depende de la interfaz implementada; no influye directamente en la arquitectura.	JSF mejora la usabilidad mediante componentes reutilizables y una interfaz web más consistente.	No aplica directamente; la usabilidad depende de la capa de presentación (JSF).	No aplica directamente; es una base de datos, su usabilidad depende de las herramientas de administración utilizadas.
Confiabilidad	Menor confiabilidad en sistemas grandes debido al riesgo de fallos globales. Un error afecta a toda la aplicación.	Proporciona confiabilidad al distribuir responsabilidades entre capas y manejar transacciones robustas.	Alta confiabilidad gracias al soporte de clustering, failover y monitoreo en Payara.	Elevada confiabilidad con replicación, clustering y recuperación ante fallos.
Seguridad	Difícil de mantener en sistemas grandes; requiere implementación manual de autenticación y	Jakarta EE incluye autenticación, autorización y manejo de sesiones seguras.	Ofrece características avanzadas de seguridad y cifrado configurable.	Proporciona cifrado de datos, autenticación robusta y control de acceso basado en roles.

Atributo de Calidad	Arquitectura Monolítica (Capas Monolito)	Jakarta EE + JSF	GlassFish / Payara	MariaDB
	control de acceso.			
Mantenibilidad	Baja mantenibilidad en aplicaciones extensas; el código tiende a acoplarse.	Alta mantenibilidad gracias a su modularidad (separación por capas).	Facilita el mantenimiento mediante herramientas de monitoreo y despliegue.	Mantenimiento sencillo en entornos pequeños, pero puede requerir experiencia técnica en clusters y replicación.
Portabilidad	Limitada, ya que depende del entorno del servidor monolítico.	Alta portabilidad dentro del ecosistema Java (corre en cualquier servidor compatible con Jakarta EE).	Portabilidad alta entre servidores Java compatibles (Payara, GlassFish, WildFly).	Portabilidad amplia entre sistemas operativos y entornos de desarrollo, compatible con MySQL.

Matriz de análisis de Tácticas vs Temas

Las tácticas de diseño son enfoques prácticos para resolver problemas técnicos en el desarrollo de software. Estas tácticas ayudan a los desarrolladores a abordar desafíos específicos relacionados con el rendimiento, la seguridad, la escalabilidad y la fiabilidad de las aplicaciones. Las tácticas se basan en experiencias previas y en el uso de patrones de diseño establecidos para implementar soluciones probadas y confiables a problemas recurrentes en el software. A continuación, se presenta una matriz que analiza las tácticas de diseño en relación con los temas asignados:

Táctica de Diseño	Arquitectura Monolítica (Capas Monolito)	Jakarta EE + JSF	GlassFish/Payara	MariaDB
Modularización	Limitada a las capas internas del monolito.	Jakarta EE promueve la modularización con EJB y JPA.	Payara soporta la modularización con microservicios.	MariaDB permite la modularización mediante bases de datos distribuidas.
Desacoplamiento	Alto acoplamiento	JSF permite desacoplar la	Soporta desacoplamiento	No aplica directamente, es

Táctica de Diseño	Arquitectura Monolítica (Capas Monolito)	Jakarta EE + JSF	GlassFish/Payara	MariaDB
	entre capas y componentes.	interfaz de usuario de la lógica de negocio.	mediante inyección de dependencias.	una base de datos.
Optimización	Requiere esfuerzos adicionales para optimizar el rendimiento.	Jakarta EE permite la optimización mediante configuración de recursos.	Optimización de procesos transaccionales.	Optimización en consultas y operaciones de base de datos.

Matriz de análisis de Patrones vs Temas

Los patrones de diseño son soluciones reutilizables a problemas comunes que los desarrolladores enfrentan al construir software. Estos patrones ofrecen un enfoque estandarizado para resolver problemas que se presentan regularmente en el desarrollo de sistemas, como el acoplamiento de componentes, la gestión de dependencias o la interacción entre capas. Algunos de los patrones más conocidos incluyen el MVC (Modelo-Vista-Controlador), el Singleton, el Factory, entre otros, los cuales ayudan a estructurar y organizar mejor el código y hacen más fácil la mantenibilidad y la extensibilidad del software. A continuación, se presenta una matriz que analiza los patrones más relevantes en relación con las tecnologías asignadas:

Patrón de Diseño	Arquitectura Monolítica (Capas Monolito)	Jakarta EE + JSF	GlassFish/Payara	MariaDB
MVC (Modelo-Vista-Controlador)	Muy adecuado para organizar las capas.	JSF implementa MVC para separar lógica de presentación y negocio.	Facilita la implementación de MVC.	No aplica directamente.
Singleton	Usado en servicios globales dentro de la aplicación.	Jakarta EE soporta el patrón Singleton a través de EJB.	GlassFish y Payara soportan singleton para servicios globales.	No aplica directamente.
Factory	Se utiliza para crear instancias de objetos dentro del monolito.	Jakarta EE facilita la creación de objetos mediante inyección de dependencias.	Payara soporta patrones como Factory a través de CDI.	No aplica directamente.

Matriz de análisis de Mercado Laboral vs Temas

El mercado laboral está directamente influenciado por la demanda de tecnologías en el desarrollo de software. Las tendencias del mercado determinan qué tecnologías son más solicitadas por las empresas y qué habilidades son más valoradas en los desarrolladores. En los últimos años, ha habido una creciente demanda de arquitecturas basadas en microservicios, pero Jakarta EE, JSF, Payara, GlassFish y MariaDB siguen siendo tecnologías clave en el mercado empresarial debido a su robustez, escala y flexibilidad en aplicaciones empresariales. Los desarrolladores con experiencia en este stack siguen siendo muy demandados por grandes empresas que priorizan la estabilidad y seguridad en sus sistemas. A continuación, se presenta una matriz que analiza cómo los temas asignados se alinean con las tendencias actuales del mercado laboral:

Tecnología	Demanda en el Mercado Laboral
Arquitectura Monolítica	Aunque las arquitecturas de microservicios están en auge, la arquitectura monolítica sigue siendo común, especialmente en empresas que ya utilizan sistemas grandes y tradicionales.
Jakarta EE + JSF	Alta demanda en empresas que requieren aplicaciones empresariales robustas. A menudo se busca experiencia en Jakarta EE para el desarrollo de aplicaciones de gran escala.
GlassFish/Payara	Demandados en entornos empresariales que requieren alta disponibilidad y transacciones seguras. Payara, en particular, está siendo adoptado por organizaciones que buscan soporte extendido.
MariaDB	Creciente demanda debido a su popularidad en aplicaciones de bases de datos web. Empresas que usan MySQL están migrando hacia MariaDB por su estabilidad y características avanzadas.

Ejemplo Práctico

El ejemplo práctico desarrollado para este trabajo corresponde a un Sistema de Gestión de Reservas de Hotel, diseñado e implementado bajo el estilo arquitectónico Capas Monolito. Este sistema tiene como objetivo principal permitir la gestión eficiente de las reservas y de los clientes dentro de un entorno hotelero, proporcionando una interfaz web sencilla, organizada y funcional, donde tanto los usuarios internos (recepcionistas) como los clientes pueden realizar operaciones básicas de registro y administración de reservas.

La solución se estructura en una sola aplicación monolítica desplegada en un servidor de aplicaciones Payara/GlassFish, y conectada a una única base de datos MariaDB, en la que se almacenan las entidades principales del dominio: Cliente y Reserva. Esta arquitectura sigue el principio de una sola base de datos compartida, lo que facilita el mantenimiento, la coherencia de los datos y el despliegue centralizado del sistema.

El sistema permite ejecutar operaciones **CRUD** (Crear, Leer, Actualizar y Eliminar) sobre las dos entidades mencionadas:

- **Cliente:** Contiene la información personal de cada usuario, como nombre, correo electrónico y teléfono.
- **Reserva:** Registra los datos asociados a una reserva, incluyendo las fechas de inicio y fin, el número de habitación y la relación con un cliente determinado.

La interacción con el sistema se realiza a través de una interfaz web construida con JavaServer Faces (JSF), donde los formularios (`.xhtml`) permiten a los usuarios ingresar, consultar o modificar la información de clientes y reservas. Estas vistas se comunican con Managed Beans (controladores) encargados de procesar las solicitudes y conectarse con la capa de Servicios, la cual implementa las reglas de negocio y la lógica transaccional. Finalmente, la capa de persistencia, desarrollada mediante Jakarta Persistence API (JPA), se encarga de mapear las entidades hacia la base de datos MariaDB, gestionando todas las operaciones de almacenamiento y consulta a través del `EntityManager`.

Desde una perspectiva técnica, el sistema mantiene una clara separación de responsabilidades entre las capas, lo que facilita su comprensión y mantenimiento:

- **Capa de Presentación:** Implementada con JSF y páginas `.xhtml`, ofrece la interfaz gráfica para la interacción con el usuario.
- **Capa de Controladores:** Incluye los Managed Beans que actúan como puente entre la presentación y la lógica de negocio.
- **Capa de Servicios:** Contiene las reglas de negocio, la validación de datos y la orquestación de las transacciones.
- **Capa de Persistencia:** Gestiona el acceso a la base de datos mediante repositorios JPA y las entidades `Cliente` y `Reserva`.

El flujo típico de una operación, como la creación de una reserva, inicia cuando el usuario completa un formulario web y lo envía. La solicitud viaja desde la capa de presentación hasta el controlador `ReservaBean`, que invoca a `ReservaService` para validar la información y registrar la transacción. Este servicio delega la persistencia al `ReservaRepository`, el cual utiliza el `EntityManager` para realizar la inserción en la base de datos. Finalmente, el resultado se propaga de vuelta a la interfaz, donde el usuario recibe la confirmación de la operación.

Diagrama de Alto nivel

El Diagrama de Alto Nivel (HDL) representa la visión general del sistema, mostrando cómo se organiza la aplicación dentro de su arquitectura y cómo interactúan los principales componentes que la conforman. Este tipo de diagrama permite comprender de manera clara y visual la estructura

del sistema sin entrar en detalles técnicos de implementación, identificando las capas principales, sus responsabilidades y la forma en que se comunican entre sí. En el caso del Sistema de Gestión de Reservas de Hotel, el diagrama de alto nivel se construye siguiendo la arquitectura en Capas Monolito, donde toda la funcionalidad de la aplicación se encuentra centralizada en un único despliegue que integra las capas de presentación, lógica de negocio y persistencia, conectadas a una única base de datos.

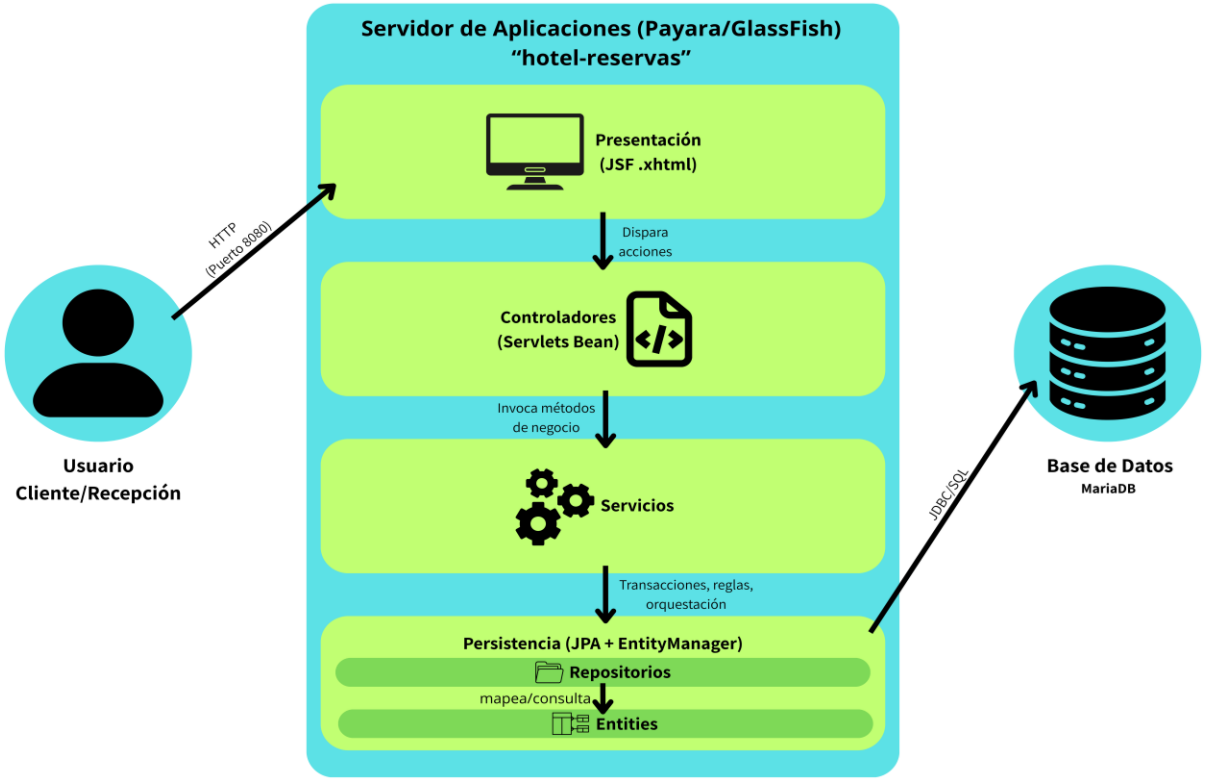


Imagen 1. Diagrama de Alto nivel de las Reservas de Hotel

El diagrama presentado ilustra cómo el usuario (que puede ser un cliente o un empleado de recepción) interactúa con el sistema a través de una interfaz web. Esta interfaz se ejecuta dentro de un servidor de aplicaciones (Payara o GlassFish), donde reside la aplicación monolítica denominada *"hotel-reservas"*.

En la parte superior del diagrama se ubica la Capa de Presentación (JSF .xhtml), encargada de mostrar las vistas y formularios donde el usuario ingresa la información de las reservas y los clientes. Desde esta capa, se disparan acciones que son procesadas por la siguiente capa: los Controladores (Servlets Bean). Estos actúan como intermediarios, gestionando las solicitudes provenientes de la interfaz y comunicándose con la Capa de Servicios, encargada de aplicar las reglas de negocio, validar la información y ejecutar las transacciones necesarias.

Posteriormente, la Capa de Persistencia (JPA + EntityManager) se encarga de realizar la conexión con la base de datos MariaDB, gestionando las operaciones de almacenamiento, actualización, consulta y eliminación de datos. Dentro de esta capa se encuentran los Repositorios, que realizan las operaciones CRUD sobre las entidades, y las Entities, que representan las tablas de la base de datos (`Cliente` y `Reserva`). La comunicación entre la aplicación y la base de datos se realiza mediante JDBC/SQL, asegurando la persistencia de los datos de manera eficiente y coherente.

Modelo C4

El Modelo C4 es una metodología de representación arquitectónica que permite describir un sistema de software desde diferentes niveles de detalle, facilitando su comprensión tanto para perfiles técnicos como no técnicos. El nombre "C4" proviene de sus cuatro niveles de abstracción: Contexto**,** Contenedores**,** Componentes **y** Código (Clases)**.** Cada nivel ofrece una vista complementaria del sistema, desde su relación con los actores externos hasta la estructura interna de sus elementos y su implementación final.

En el caso del Sistema de Gestión de Reservas de Hotel, el modelo C4 se utiliza para representar de forma estructurada la arquitectura en Capas Monolito implementada con Jakarta EE, JSF y MariaDB. Cada nivel del modelo describe un aspecto diferente del sistema:

- El **Diagrama de Contexto** muestra la interacción entre los usuarios y el sistema en su entorno general.
- El **Diagrama de Contenedores** detalla cómo se organiza internamente la aplicación en términos de módulos y base de datos.
- El **Diagrama de Componentes** descompone la aplicación en sus principales elementos funcionales (presentación, controladores, servicios y persistencia).
- Finalmente, el **Diagrama de Código** profundiza en la implementación de clases y sus relaciones dentro del sistema.

Diagrama de Contexto

El Diagrama de Contexto corresponde al primer nivel del modelo C4, y tiene como propósito ofrecer una visión general del sistema dentro de su entorno, mostrando su relación con los actores externos que interactúan con él. Este diagrama define los límites del sistema, los tipos de usuarios que lo utilizan y las interacciones principales que se llevan a cabo. De esta manera, permite comprender de forma global qué hace el sistema**,** quién lo utiliza **y** qué valor aporta dentro de su contexto de uso, sin entrar aún en detalles sobre su estructura interna o los componentes que lo conforman.

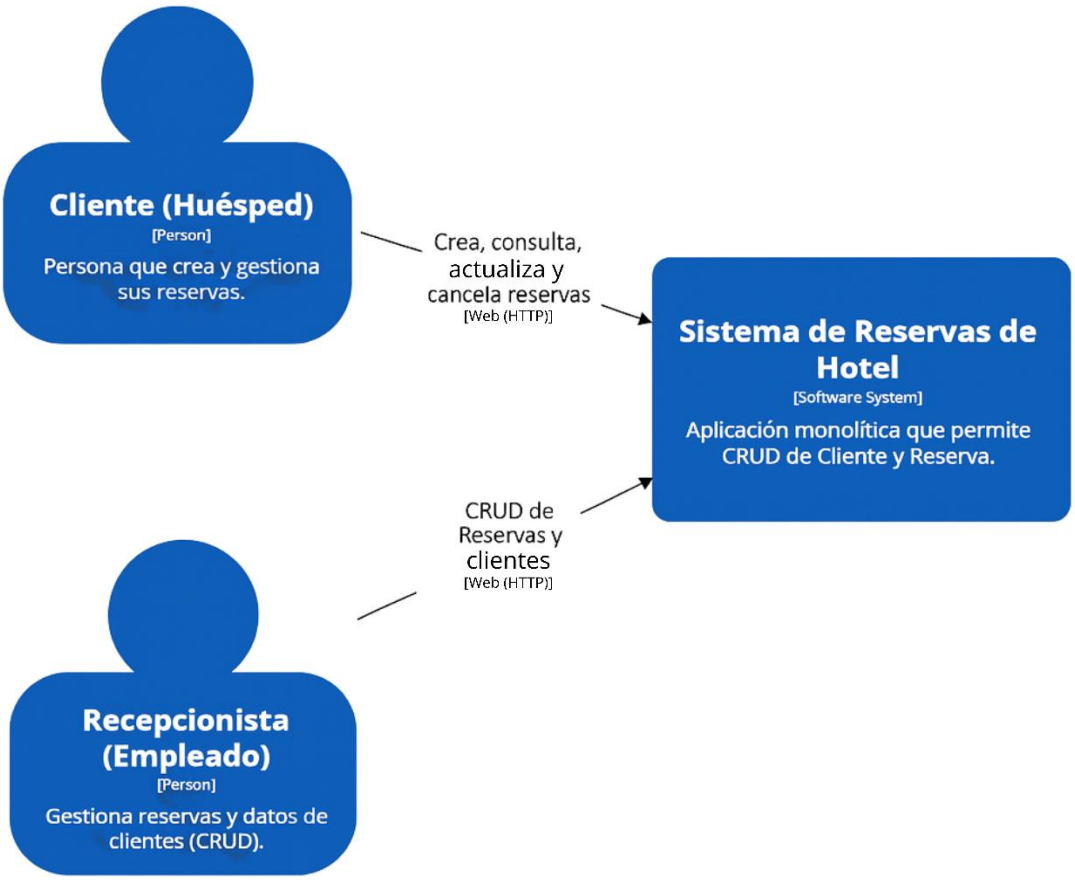


Imagen 2. Diagrama de Contexto de las reservas de Hotel.

En el caso del Sistema de Gestión de Reservas de Hotel, el diagrama de contexto ilustra cómo el sistema centraliza todas las operaciones de reserva y gestión de clientes dentro de una aplicación monolítica, accesible mediante un navegador web. Se identifican dos actores principales que interactúan con el sistema:

- **Cliente (Huésped):** Es la persona que utiliza la aplicación para crear, consultar, actualizar o cancelar sus reservas de manera sencilla a través de una interfaz web. Representa el usuario final del sistema, que busca realizar operaciones relacionadas con su estancia en el hotel.
- **Recepcionista (Empleado):** Es el usuario interno del hotel encargado de gestionar las reservas y los datos de los clientes, realizando las operaciones de creación, modificación, consulta y eliminación (CRUD) de información dentro del sistema.

Ambos actores se comunican con el Sistema de Reservas de Hotel, que permite realizar todas las operaciones necesarias para el manejo de clientes y reservas. La comunicación entre los actores y el sistema se realiza mediante el protocolo **Web (HTTP)**, lo que permite su acceso desde cualquier navegador o red interna del hotel.

Diagrama de Contenedores

El Diagrama de Contenedores corresponde al segundo nivel del modelo C4 y tiene como objetivo representar la estructura interna del sistema en términos de sus principales contenedores o unidades de ejecución. Cada contenedor representa una aplicación, base de datos, microservicio, API o cualquier elemento que pueda ejecutarse y cumplir un rol funcional dentro del sistema. Este nivel de detalle permite comprender cómo se distribuye la funcionalidad dentro del sistema**,** qué tecnologías utiliza cada componente principal **y** cómo se comunican entre sí.

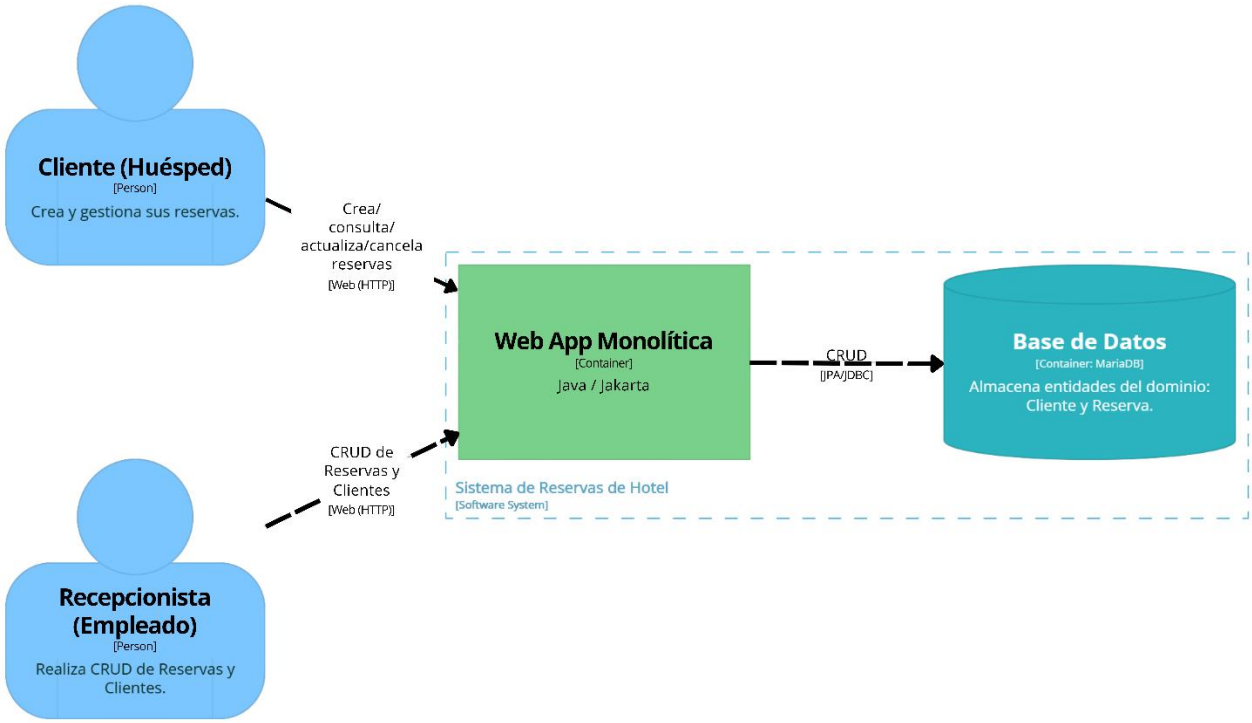


Imagen 3. Diagrama de Contenedores de las reservas de Hotel.

En el caso del Sistema de Gestión de Reservas de Hotel, el diagrama de contenedores muestra una arquitectura monolítica por capas, compuesta por una única aplicación web desplegada en un servidor de aplicaciones Java y una base de datos relacional MariaDB. Ambos conforman el núcleo funcional del sistema, que se ejecuta de manera centralizada, garantizando coherencia, simplicidad de despliegue y facilidad de mantenimiento.

El contenedor principal es la Web App Monolítica. Este componente concentra toda la lógica de negocio, la presentación de la interfaz de usuario y las operaciones de persistencia de datos.

Diagrama de Componentes

El Diagrama de Componentes corresponde al tercer nivel del modelo C4, y tiene como objetivo mostrar cómo se estructura internamente cada contenedor del sistema, evidenciando los componentes de software que lo conforman, sus responsabilidades y las interacciones entre ellos.

Este nivel de detalle permite entender cómo se distribuye la lógica de negocio dentro de la aplicación **y** de qué manera se comunican las distintas capas que conforman la arquitectura.

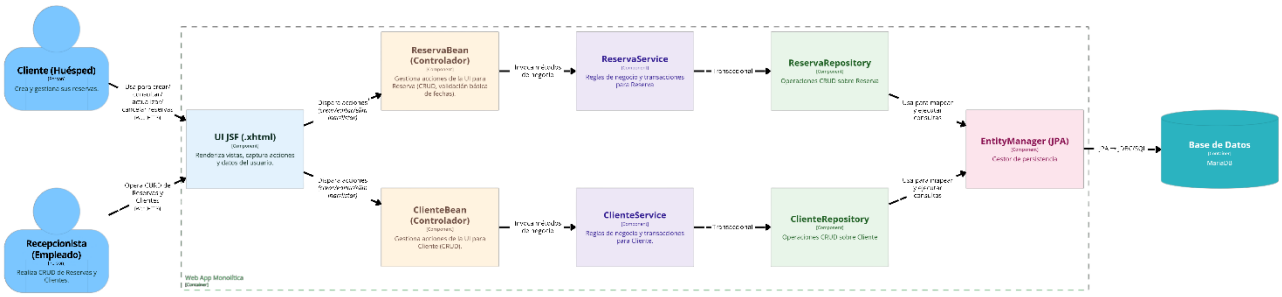


Imagen 4. Diagrama de Componentes de las reservas de Hotel.

En el caso del Sistema de Gestión de Reservas de Hotel, el diagrama de componentes detalla la estructura interna de la Web App Monolítica. Esta aplicación contiene los módulos necesarios para gestionar las entidades principales del dominio (Cliente **y** Reserva), abarcando desde la interfaz de usuario hasta la persistencia de datos en la base de datos MariaDB.

El diagrama se organiza de izquierda a derecha siguiendo el flujo de interacción del sistema:

- En el extremo izquierdo se ubican los **usuarios externos**: el Cliente (Huésped) y el Recepcionista (Empleado), quienes interactúan con la aplicación a través de la web.
- Ambos acceden a la Interfaz de Usuario (UI JSF .xhtml)**,** que corresponde a la capa de presentación. Este componente se encarga de renderizar las vistas, capturar acciones y recibir la información ingresada por los usuarios a través de formularios web.

Desde la interfaz, las acciones son enviadas a los controladores, representados por los componentes ReservaBean y ClienteBean. Estos Managed Beans (Servlets Bean) son responsables de procesar las solicitudes, aplicar validaciones básicas y coordinar la ejecución de las operaciones CRUD correspondientes.

Cada controlador invoca los métodos de su respectivo servicio de negocio, definidos en los componentes ReservaService y ClienteService. En esta capa se concentran las reglas de negocio, las transacciones y la lógica de validación que garantiza la coherencia de los datos. Los servicios actúan de manera transaccional, asegurando que las operaciones se ejecuten correctamente en la base de datos.

Luego, los servicios interactúan con la capa de persistencia, compuesta por los repositorios ReservaRepository y ClienteRepository, para realizar las operaciones CRUD sobre las entidades del dominio. El `EntityManager` actúa como gestor de persistencia, encargándose de mapear los objetos Java a las tablas de la base de datos mediante JPA (Jakarta Persistence API) y ejecutar las operaciones SQL correspondientes a través de JDBC.

Finalmente, **la** Base de Datos MariaDB almacena de manera permanente las entidades del dominio (Cliente **y** Reserva), garantizando la integridad y disponibilidad de la información.

Diagrama de Código

El Diagrama de Código corresponde al cuarto y último nivel del modelo C4, y tiene como finalidad mostrar la estructura interna del sistema a nivel de implementación. Este nivel representa cómo se organizan los paquetes, clases, componentes de configuración y entidades dentro del código fuente, evidenciando las relaciones entre ellos, los patrones de diseño utilizados y la manera en que las dependencias se integran para ejecutar las operaciones del sistema.

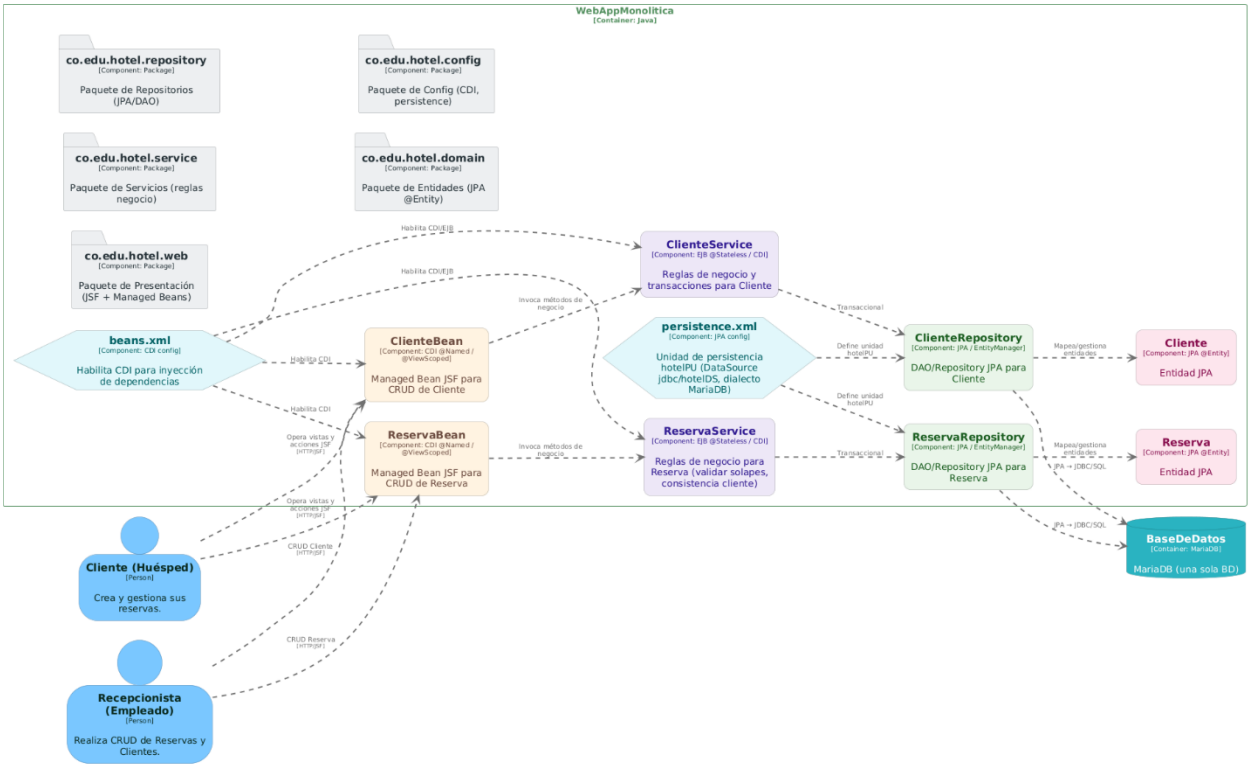


Imagen 5. Diagrama de Código de las reservas de Hotel.

En el caso del Sistema de Gestión de Reservas de Hotel, el diagrama de código detalla la organización interna de la aplicación WebAppMonolítica. Este diagrama refleja el conjunto de paquetes, clases y configuraciones que hacen posible la implementación del sistema, desde la capa de presentación hasta la persistencia de datos en la base de datos MariaDB.

En la parte superior se presentan los principales paquetes del sistema, los cuales se alinean con las capas de la arquitectura:

- **`co.edu.hotel.web`**: Contiene los componentes de la capa de presentación, donde se implementan los Managed Beans asociados a las páginas.
- **`co.edu.hotel.service`**: Agrupa la lógica de negocio y las transacciones del sistema, implementadas a través de clases de servicio (EJB o CDI).

- **`co.edu.hotel.repository`**: Incluye los ***repositorios JPA (DAO*)** encargados de realizar las operaciones CRUD sobre las entidades.
- **`co.edu.hotel.domain`**: Define las entidades del dominio (`Cliente` y `Reserva`), mapeadas a las tablas de la base de datos mediante las anotaciones @Entity **y** @Id.
- **`co.edu.hotel.config`**: Contiene los archivos de configuración que habilitan los servicios de CDI (Context and Dependency Injection) y JPA (Jakarta Persistence), garantizando la conexión con la base de datos y la inyección de dependencias entre componentes.

En la capa de presentación se destacan los componentes **`ClienteBean`** y **`ReservaBean`**, que funcionan como controladores, con las anotaciones @Named y @ViewScoped, permitiendo gestionar las operaciones CRUD de cada entidad directamente desde la interfaz web. Estos controladores se comunican con los servicios correspondientes (**`ClienteService`** y **`ReservaService`**), los cuales están anotados con @Stateless y @Inject para indicar su naturaleza transaccional y su disponibilidad mediante inyección de dependencias.

Los servicios de negocio, a su vez, interactúan con los repositorios JPA (**`ClienteRepository`** y **`ReservaRepository`**), que implementan el patrón DAO (Data Access Object) utilizando el EntityManager proporcionado por la unidad de persistencia configurada en el archivo **`persistence.xml`**. Este archivo define la conexión con la base de datos MariaDB, el dialecto SQL utilizado y el `DataSource` correspondiente (`jdbc/hotelDS`).

Finalmente, las clases **`Cliente`** y **`Reserva`** representan las entidades del dominio, anotadas con @Entity, @Table y los campos con @Column y @GeneratedValue, mapeando directamente los atributos del modelo a las columnas de la base de datos.

Diagrama Dinámico C4

El Diagrama Dinámico corresponde a una extensión del modelo C4 que permite representar el flujo secuencial de interacciones entre los diferentes componentes del sistema durante la ejecución de un proceso o caso de uso específico. A diferencia de los diagramas anteriores que se enfocan en la estructura estática del sistema, este diagrama resalta la dinámica de comunicación y los mensajes que se intercambian entre los elementos del sistema para completar una acción concreta.

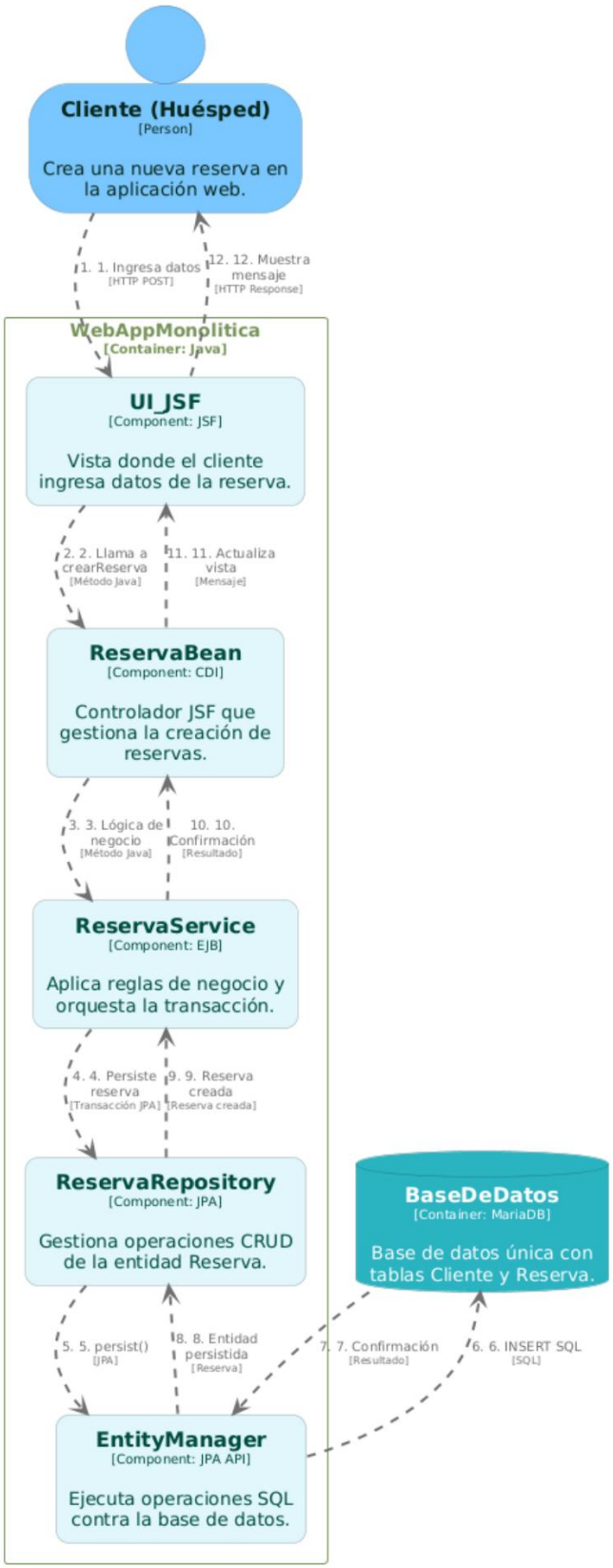


Imagen 6. Diagrama Dinámico de las reservas de Hotel.

En el Sistema de Gestión de Reservas de Hotel, el diagrama dinámico describe el flujo completo del proceso de creación de una nueva reserva por parte de un cliente (huésped) desde la interfaz web. Este proceso ejemplifica cómo los distintos componentes de la aplicación monolítica colaboran para ejecutar una transacción de negocio.

El flujo se desarrolla de la siguiente manera:

1. El Cliente (Huésped) accede a la aplicación web y completa el formulario de reserva en la interfaz UI JSF, enviando los datos mediante una petición HTTP POST.
2. La vista JSF (.xhtml) captura la información ingresada y llama al método `crearReserva()` del controlador ReservaBean.
3. El controlador ReservaBean, aplica validaciones iniciales (como verificar las fechas de inicio y fin) e invoca la lógica de negocio en el componente ReservaService.
4. El servicio ReservaService, coordina la creación de la reserva y delega la persistencia al componente ReservaRepository.
5. El repositorio ReservaRepository, utiliza el EntityManager (JPA) para ejecutar la operación `persist()` y almacenar la nueva reserva en la base de datos.
6. El EntityManager transforma la entidad `Reserva` en una instrucción SQL INSERT, que se ejecuta sobre la Base de Datos MariaDB, donde se almacena la información en la tabla correspondiente.
7. La base de datos confirma la operación y devuelve una respuesta de éxito al EntityManager, indicando que la reserva ha sido registrada correctamente.
8. El EntityManager devuelve la entidad persistida al ReservaRepository, que la propaga de regreso al ReservaService.
9. El servicio confirma la transacción y notifica al controlador ReservaBean el resultado exitoso.
10. El controlador actualiza el estado de la vista y prepara un mensaje de confirmación.
11. La vista JSF se actualiza, mostrando al cliente un mensaje de éxito en pantalla.
12. Finalmente, el cliente visualiza la confirmación de que su reserva ha sido creada correctamente, completando así el ciclo de interacción.

Diagrama de Despliegue C4

El Diagrama de Despliegue constituye una extensión del modelo C4, y su objetivo principal es mostrar cómo los contenedores y componentes del sistema se distribuyen en el entorno físico o virtual donde se ejecutan. Este diagrama permite comprender la infraestructura de ejecución, los nodos de despliegue (servidores, máquinas virtuales o contenedores), así como las conexiones de red y los protocolos de comunicación utilizados entre los distintos elementos.

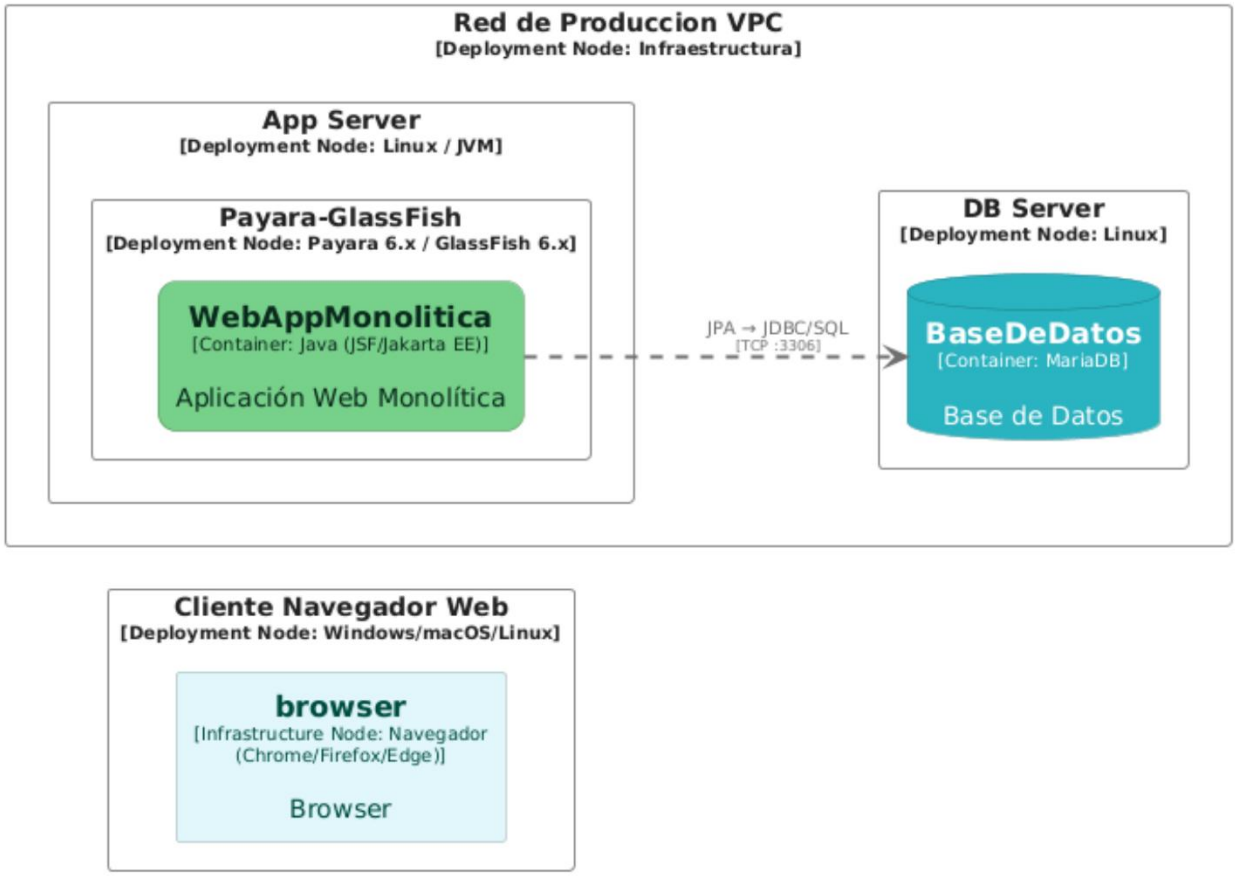


Imagen 7. Diagrama de Despliegue de las reservas de Hotel.

En el caso del Sistema de Gestión de Reservas de Hotel, el diagrama de despliegue representa la infraestructura técnica necesaria para la ejecución de la aplicación monolítica, destacando la integración entre el servidor de aplicaciones, el servidor de base de datos y el cliente web que accede al sistema.

La WebAppMonolítica, se despliega en un servidor de aplicaciones, alojado dentro de un nodo denominado App Server. Este servidor se ejecuta sobre un entorno Linux con JVM (Java Virtual Machine), lo que permite la ejecución de la aplicación empaquetada en formato `.war` bajo un contexto llamado `hotel-reservas`. Dentro del mismo entorno, el servidor de aplicaciones se encarga de gestionar los servicios de Jakarta EE, la inyección de dependencias (CDI) y la comunicación con la capa de persistencia (JPA).

Por otro lado, el Servidor de Base de Datos (DB Server), también basado en Linux, alberga una instancia de MariaDB. En esta base de datos se almacenan las entidades principales del dominio (Cliente y Reserva), gestionadas a través de JPA (Jakarta Persistence API) y conectadas mediante el protocolo JDBC/SQL (puerto 3306). Esta conexión permite que la aplicación persista los datos, ejecute transacciones y realice las operaciones CRUD correspondientes de manera confiable.

El sistema se encuentra desplegado dentro de una Red de Producción (VPC), la cual garantiza la comunicación interna segura entre los servidores de aplicación y base de datos. Este entorno puede

ejecutarse tanto en infraestructura local (on-premise) como en entornos virtualizados o en la nube, manteniendo la arquitectura monolítica centralizada.

Finalmente, los usuarios del sistema (Cliente y Recepcionista) acceden a la aplicación a través de un navegador web (Browser), que puede ejecutarse en cualquier sistema operativo (Windows, macOS o Linux). La comunicación entre el cliente y la aplicación se realiza mediante el protocolo HTTP (puerto 8080), lo que permite acceder a las funcionalidades del sistema desde cualquier dispositivo conectado a la red.

Diagrama de Paquetes UML

El Diagrama de Paquetes UML es una herramienta de modelado estructural que permite representar la organización lógica del código fuente dentro de un sistema, mostrando cómo se agrupan las clases, componentes y módulos en distintos paquetes, así como las dependencias existentes entre ellos. Este tipo de diagrama resulta fundamental para comprender la arquitectura interna de una aplicación, ya que facilita la visualización de la modularidad, la separación de responsabilidades y la interrelación entre las capas del sistema.

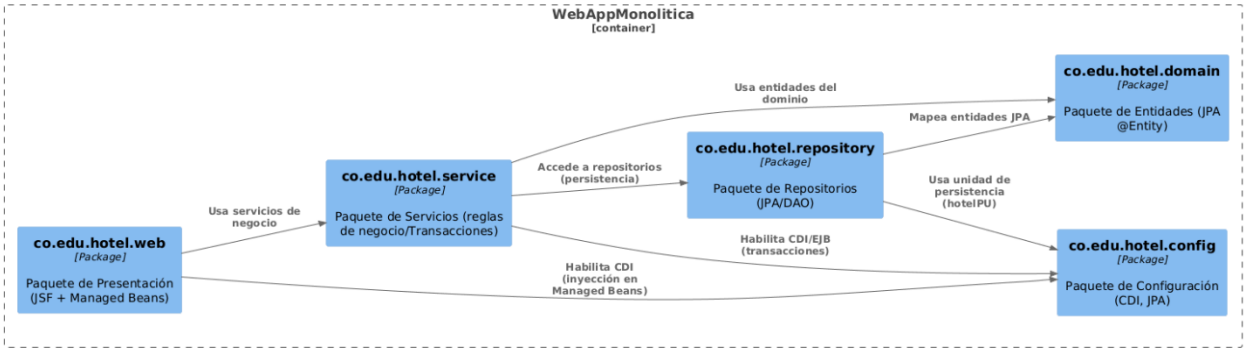


Imagen 8. Diagrama de Paquetes UML de las reservas de Hotel

En el caso del Sistema de Gestión de Reservas de Hotel, el Diagrama de Paquetes UML representa la organización lógica del código fuente de la aplicación, siguiendo una estructura modular basada en la arquitectura en capas. Este diagrama permite visualizar cómo se agrupan las clases en diferentes paquetes según sus responsabilidades, así como las dependencias existentes entre las capas del sistema, garantizando una visión clara de la modularidad, la cohesión interna y la mantenibilidad del software.

El diagrama se encuentra dentro del contenedor WebAppMonolítica, que encapsula los principales paquetes del sistema:

- **`co.edu.hotel.web` (Paquete de Presentación):** Incluye los componentes de la interfaz de usuario, que se encargan de capturar las acciones del usuario y comunicarse con la capa de servicios para procesar las operaciones CRUD sobre clientes y reservas.
- **`co.edu.hotel.service` (Paquete de Servicios):** Contiene la lógica de negocio y la gestión de transacciones. Aquí se definen las clases `ClienteService` y `ReservaService`,

responsables de aplicar las reglas del negocio, validar la información y orquestar las operaciones necesarias antes de acceder a la capa de persistencia.

- **`co.edu.hotel.repository` (Paquete de Repositorios):** Implementa el acceso a la base de datos mediante JPA (Jakarta Persistence API) bajo el patrón DAO (Data Access Object)**.** Los repositorios, como `ClienteRepository` y `ReservaRepository`, se comunican con el `EntityManager` configurado para ejecutar operaciones CRUD sobre las entidades del dominio.
- **`co.edu.hotel.domain` (Paquete de Entidades):** Define las entidades principales del sistema, `Cliente` y `Reserva`, mapeadas a las tablas de la base de datos con anotaciones @Entity. Estas clases representan el modelo de datos y son utilizadas por las capas de servicio y repositorio para gestionar la información persistente.
- **`co.edu.hotel.config` (Paquete de Configuración):** Agrupa los archivos y clases que permiten la configuración del sistema, tales como **`beans.xml`** (que habilita CDI para la inyección de dependencias) y **`persistence.xml`** (que define la unidad de persistencia **`hotelPU`**, el DataSource **`jdbc/hotelDS`** y el dialecto de MariaDB utilizado por JPA).

Las relaciones mostradas en el diagrama reflejan las dependencias lógicas y direccionales entre los paquetes:

- El paquete **`web`** depende de **`service`** para utilizar la lógica de negocio.
- El paquete **`service`** depende de **`repository`** para realizar operaciones de persistencia y de **`domain`** para manipular las entidades del modelo.
- Los paquetes **`web`**, **`service`** y **`repository`** dependen de **`config`**, que habilita CDI y define la configuración JPA necesaria para el funcionamiento del sistema.

Conclusiones

El desarrollo del Sistema de Gestión de Reservas de Hotel permitió integrar y comprender de forma práctica los conceptos fundamentales asociados a la arquitectura de Capas Monolito, utilizando tecnologías del ecosistema Jakarta EE. A través de la implementación de JSF, EJB/CDI, y JPA, se logró estructurar una aplicación modular, organizada y con separación clara entre la presentación, la lógica de negocio y la persistencia.

Asimismo, el uso de Payara como servidor de aplicaciones facilitó el despliegue y la gestión de componentes Java empresariales, mientras que MariaDB proporcionó un entorno de base de datos relacional robusto, seguro y de alto rendimiento.

Este trabajo evidenció que, aunque las arquitecturas monolíticas presentan limitaciones frente a modelos más modernos como los microservicios, siguen siendo una opción viable, eficiente y mantenible para sistemas de tamaño pequeño o mediano que priorizan la simplicidad, consistencia y rapidez de desarrollo. En conjunto, las tecnologías estudiadas demuestran su vigencia y aplicabilidad en contextos empresariales donde la estabilidad y la integración son factores clave.

Referencias:

1. Bass, L., Clements, P., & Kazman, R. (2013). *Software Architecture in Practice* (3rd ed.). Addison-Wesley.
2. Garlan, D., & Shaw, M. (1994). *An Introduction to Software Architecture*. In *Advances in Software Engineering and Knowledge Engineering* (pp. 1-39). World Scientific Publishing.
3. Richards, M. (2015). *Microservices vs Monolithic Architectures*. O'Reilly Media.
4. Goncalves, A. (2020). *Beginning Jakarta EE: Enterprise Edition for Java* (2nd ed.). Apress.
5. Pau, L. (2016). *JavaServer Faces (JSF) 2.0: The Complete Reference*. McGraw-Hill Education.
6. Moris, R. (2018). *Mastering Jakarta EE: Building Scalable and Secure Applications with Java*. Packt Publishing.
7. Payara. (2021). *What is Payara?*. https://www.payara.fish/
8. Goncalves, A. (2020). *Beginning Jakarta EE: Enterprise Edition for Java* (2nd ed.). Apress.
9. Moris, R. (2018). *Mastering Jakarta EE: Building Scalable and Secure Applications with Java*. Packt Publishing.
10. Widenius, M. (2018). *MariaDB: The Complete Reference*. McGraw-Hill Education.
11. MariaDB Foundation. (2021). *MariaDB Overview*. https://mariadb.org/
12. Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.
13. Bass, L., Clements, P., & Kazman, R. (2013). *Software Architecture in Practice* (3rd ed.). Addison-Wesley.
14. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
15. Dahanayake, A., & Sol, H. (2012). *The Importance of Java EE and Its Influence on Software Development Careers*. Journal of Software Engineering.