12/08/25 – BACKEND DE APLICACIONES

Capas de software que se comunican entre sí: cliente, frontend, backend o database en un modelo simplificado. Cada capa está asociada a una tecnología y en base a ella se construye la aplicación. Se puede decir que el Stack es la pila de capas que se comunican con la anterior y la siguiente; entonces es también la pila de tecnologías con las que se va a construir la aplicación. Donde Frontend es la capa donde se programa la IU, Backend la capa donde se programa la lógica de negocio y por arriba los consumidores (Clientes) y por debajo la persistencia de los datos (DBR).

* Cliente: todo dispositivo al que se conecta la aplicación (browser/ejecutables). Se trata del entorno de ejecución de la interfaz de usuario.
* Frontend: donde se desarrolla la presentación y estilo visual de la información para el usuario. Contiene la captura de interacción y la lógica asociada a los procesos de captura y presentación que correspondan con los requerimientos planteados por nuestra aplicación.
* Backend: capa donde se motorizan las reglas de negocio que debemos validar ante las entidades de sistema. Debemos involucrar técnicas de control de acceso y optimizar procesos para lograr dar servicio a la mayor cantidad de usuarios con recursos acotados.
* Database: es la capa de datos relacionada con todo lo que la aplicación deba persistir (BD, archivos o interconexión con otros sistemas).
* Herramientas: repositorio de código fuente, testing, entorno integrado para escribir código y administrar proyectos.

BACKEND: conjunto de servicios y componentes que trabajan en conjunto para dar soporte a la funcionalidad de la aplicación. Debe ser una parte robusta y segura de la arquitectura (escalable, fácil de mantener y capaz de manejar grandes volúmenes de tráfico).

Desafíos:

* Manejo de la lógica de negocio: reglas y procesos que definen como funciona el sistema y como se llevan a cabo las operaciones. Fundamental para que la aplicación cumpla con los objetivos y funcione correctamente. Comprensión clara de los requerimientos de negocio y habilidad para traducirlos a código funcional.
* Escalabilidad: capacidad del backend para manejar un aumento en la carga de trabajo o el número de usuarios sin degradar el rendimiento o la disponibilidad del sistema.
* Seguridad: implica proteger los datos y recursos del sistema de accesos no autorizados y ataques maliciosos. Proteger la información sensible de los usuarios y garantizar la integridad de los datos.
* Rendimiento: capacidad del backend para responder rápidamente a las solicitudes de los usuarios y proporcionar una experiencia fluida y ágil.
* Integración con sistemas externos: debe integrarse con sistemas externos, como servicios web, API de terceros o sistemas legacy, para proporcionar funcionalidades adicionales o acceder a datos externos.

**Backend Organización Interna**

Para organizar y mantener un código limpio y eficiente, el backend de aplicaciones se divide en módulos y subcapas internas. Cada módulo se enfoca en una funcionalidad específica de la aplicación, y dentro de cada módulo, existen subcapas que ayudan a separar diferentes responsabilidades.

Hay varias alternativas para organizar los componentes dentro del backend de aplicaciones. Algunas de las más comunes son:

Arquitectura Monolítica: todos los componentes de una aplicación se encuentran dentro de una única unidad. La aplicación es un todo cohesivo, y todos los servicios y funcionalidades están empaquetados juntos. Los componentes pueden comunicarse directamente entre sí, ya que comparten la misma base de código y recursos. Aunque es simple, puede volverse compleja a medida que la aplicación crece. Si bien existe la división de capas estas coexisten todas en un mismo empaquetado y no tengo forma de realizar un cambio en un componente si redesplegar y modificar toda la aplicación.

Microservicios: es un enfoque moderno que divide una aplicación en servicios pequeños, autónomos y especializados. Cada microservicio se enfoca en una única funcionalidad de la aplicación y puede ser desarrollado, desplegado y escalado de forma independiente. La aplicación se divide en servicios independientes y autónomos, cada uno con su propia base de código y funcionalidad. Esto facilita la escalabilidad y el despliegue.

Arquitectura Hexagonal (Ports & Adapters): se centra en separar la lógica de negocio (dominio) del acceso a datos y la interacción con otros sistemas externos (adaptadores). Los puertos son interfaces que definen cómo se comunica el núcleo de la aplicación con el mundo exterior, y los adaptadores son las implementaciones concretas de esas interfaces. Se centra en separar la lógica de negocio del acceso a datos y la interacción con otros sistemas, lo que mejora la modularidad y la testabilidad.

Arquitectura basada en Servicios (SOA): se enfoca en construir aplicaciones como un conjunto de servicios independientes que se comunican a través de protocolos estándar como HTTP y XML. Los servicios son componentes autónomos que encapsulan la lógica de negocio y pueden ser implementados en diferentes tecnologías. Los servicios son componentes independientes que se comunican entre sí mediante interfaces bien definidas. Se diferencia de la Arquitectura de Microservicios en que en la mayoría de los casos se implementa mediante un enfoque monolítico y, dentro de dicha aplicación monolítica, los servicios se vuelven mecanismos de interoperabilidad con otros sistemas o exposición de funcionalidades.

MICROSERVICIOS: adecuada para aplicaciones grandes y complejas. Cada microservicio puede ser desarrollado por equipos especializados y desplegado en diferentes máquinas o contenedores. Sus características son:

* Independencia: Cada microservicio es una unidad independiente, lo que significa que puede ser desarrollado, probado, desplegado y escalado por separado.
* Comunicación basada en APIs: Los microservicios se comunican entre sí a través de interfaces bien definidas, como APIs RESTful o colas de mensajes.
* Escalabilidad: Los microservicios permiten escalar solo los servicios que necesitan más recursos, lo que mejora el rendimiento y la eficiencia.
* Resiliencia: Si un microservicio falla, no afecta a los demás, lo que mejora la disponibilidad y la resistencia del sistema.
* Flexibilidad Tecnológica: Cada microservicio puede estar implementado con diferentes tecnologías, lo que permite elegir la mejor tecnología para cada funcionalidad.

En una arquitectura de microservicios, la aplicación se divide en varios servicios pequeños e independientes, cada uno de los cuales se enfoca en una funcionalidad específica. Cada microservicio es un contenedor autónomo que puede ser desarrollado, desplegado y escalado de forma independiente. La comunicación y la interacción entre estos servicios se realizan a través de endpoints bien definidos, que actúan como puntos de entrada para las solicitudes y respuestas.

Un microservicio es una unidad de despliegue que encapsula una parte del negocio o una funcionalidad específica de la aplicación. Cada microservicio contiene todo lo necesario para cumplir su propósito, incluyendo la lógica de negocio, el acceso a datos, las configuraciones y las dependencias. La independencia de los microservicios permite una mayor flexibilidad y escalabilidad, ya que cada uno puede ser modificado, actualizado y desplegado sin afectar a los demás.

Los endpoints son los puntos de acceso a los microservicios. Actúan como interfaces que permiten la comunicación entre los microservicios y los clientes (como aplicaciones frontend, otros microservicios o sistemas externos). Cada endpoint expone una o más operaciones que pueden ser invocadas a través de protocolos estándar, como HTTP/HTTPS, utilizando métodos como GET, POST, PUT y DELETE y brindan acceso a un recurso específico del negocio que podemos identificar de manera unívoca por la URI que define al endpoint.

La agrupación de conceptos en microservicios implica definir claramente los límites de cada servicio y las responsabilidades que cada uno debe asumir. Esto se hace con el objetivo de minimizar las dependencias y maximizar la cohesión dentro de cada microservicio.

La interacción entre microservicios se realiza principalmente a través de llamadas a los endpoints expuestos por cada servicio. Esta interacción puede ser sincrónica o asincrónica, dependiendo de los requisitos del sistema y la naturaleza de la comunicación.

* Comunicación Sincrónica: En este modelo, un microservicio realiza una solicitud a otro microservicio y espera una respuesta antes de continuar. Esto se suele implementar mediante APIs RESTful o llamadas HTTP.
* Comunicación Asincrónica: En este modelo, un microservicio envía un mensaje a otro microservicio y no espera una respuesta inmediata. Esto se suele implementar mediante colas de mensajes o sistemas de eventos.

**Principios Básicos de los Endpoints**

1. Simplicidad y Consistencia: Los endpoints deben ser simples, intuitivos y consistentes a lo largo de toda la aplicación. Esto facilita su uso y mantenimiento.

2. RESTful: Aunque no es obligatorio, es una práctica común estructurar los endpoints siguiendo los principios REST (Representational State Transfer). Esto incluye el uso de HTTP verbs (GET, POST, PUT, DELETE) y la utilización de URIs claras y significativas.

3. Versionado: Es importante versionar los endpoints para manejar cambios y actualizaciones en la API sin interrumpir el servicio para los clientes existentes.

**Lenguajes, Frameworks y Librerías**

Java: Java es un lenguaje de programación orientado a objetos, ampliamente utilizado en el desarrollo de aplicaciones empresariales. Es conocido por su portabilidad, facilidad de uso y robustez.

Spring Framework: framework de desarrollo de aplicaciones Java que proporciona una infraestructura completa para crear aplicaciones empresariales. Incluye módulos para inyección de dependencias, acceso a datos, seguridad, transacciones y más.

Hibernate: Hibernate es un framework de mapeo objeto-relacional (ORM) que facilita la interacción con bases de datos relacionales a través de objetos Java. Abstrae la capa de acceso a datos, lo que permite trabajar con objetos en lugar de SQL.

JUnit: JUnit es un framework de pruebas unitarias para Java. Permite escribir pruebas automatizadas para verificar el comportamiento correcto de las clases y métodos de la aplicación.

Mockito: Mockito es un framework de pruebas en Java que permite crear objetos simulados (mocks) para facilitar la escritura de pruebas unitarias. Ayuda a simular el comportamiento de objetos y servicios externos.

Maven: es una herramienta de automatización y gestión de proyectos utilizada principalmente en proyectos Java. Permite compilar, empaquetar, testear y desplegar aplicaciones de manera eficiente y estandarizada. Su principal ventaja radica en la gestión de dependencias externas mediante un archivo central de configuración (pom.xml), lo que facilita el trabajo colaborativo y la integración de librerías de terceros.

IDE: facilita la escritura, organización, depuración y ejecución del código. Un buen IDE ofrece herramientas como autocompletado, navegación entre clases, refactorización, integración con sistemas de control de versiones, ejecución de pruebas y soporte para la gestión de dependencias.

Una API RESTful (Representational State Transfer) es una interfaz de programación de aplicaciones que sigue los principios de REST. Permite a las aplicaciones comunicarse y transferir datos utilizando métodos HTTP estándar como GET, POST, PUT y DELETE.

Patrones:

El patrón Data Access Object (DAO) se utiliza para separar la lógica de acceso a datos de la lógica de negocio. Proporciona una capa de abstracción entre la aplicación y la base de datos.

El Patrón Repositorio es un patrón de diseño de software que se utiliza en el desarrollo de aplicaciones para separar la lógica de acceso a datos de la lógica de negocio. Proporciona una abstracción de la fuente de datos subyacente, como una base de datos, archivos o servicios web, a través de interfaces y clases de repositorio. Esto permite a los desarrolladores interactuar con los datos de manera coherente y centralizada, independientemente de cómo se almacenen o recuperen.

El patrón de diseño Singleton garantiza que una clase tenga una única instancia en todo el sistema, proporcionando un punto global de acceso a esta instancia

: El patrón Factory se utiliza para crear objetos sin exponer la lógica de creación. Proporciona una interfaz común para la creación de objetos de diferentes tipos.

El patrón Observer permite a objetos interesados suscribirse y recibir notificaciones cuando un objeto observado cambia su estado.

El patrón Gateway se utiliza para centralizar la lógica de acceso y seguridad en un punto de entrada único para los microservicios.

**JAVA**

Java nació como un lenguaje portable y orientado a objetos, ganó popularidad con los applets en la web y hoy es una tecnología robusta para el backend empresarial. Su evolución ha estado marcada por mejoras técnicas, cambios de nomenclatura y la transición a un ciclo de versiones más rápido con soporte LTS.

TS significa Long Term Support o Soporte a Largo Plazo. Es una versión “estable” que recibe actualizaciones de seguridad, correcciones de errores y mejoras críticas durante muchos años (normalmente 5 a 8 años o más).

Ediciones de Java

* Java SE (Standard Edition): La versión central de Java. Incluye el conjunto principal de herramientas y APIs para programar aplicaciones estándar, desde colecciones y manejo de bases de datos hasta redes e hilos.
* Java EE (Enterprise Edition) / Jakarta EE: Antes gestionada por Oracle, ahora es un proyecto de la Fundación Eclipse. Es la plataforma para aplicaciones empresariales, con perfiles específicos:
  + Perfil Web: Tecnologías para aplicaciones web (Servlets, JSF, JPA, CDI).
  + Plataforma Completa: Incluye todo el perfil web más tecnologías avanzadas (EJB, JMS, JTA).
  + MicroProfile: Comunidad para desarrollar microservicios ligeros con APIs para configuración, seguridad y más.
* Java ME (Micro Edition): Enfocada en dispositivos con recursos limitados, originalmente para móviles y hoy orientada a IoT.
* Java Card: Para desarrollar aplicaciones seguras en tarjetas inteligentes y dispositivos con recursos muy limitados.
* Java FX: Plataforma para construir interfaces gráficas modernas y ricas en aplicaciones de escritorio, web y móviles. Ahora es un proyecto open source bajo OpenJFX.

**Características de Java**

* **Alto Nivel:** Lenguaje legible para humanos, con muchas librerías que facilitan la programación sin preocuparse por detalles físicos del hardware.
* **Orientado a Objetos:** Java fue diseñado desde el inicio para trabajar con objetos; todo en Java es un objeto.
* **Independiente de la Plataforma:** Gracias a la JVM (Máquina Virtual de Java) para diferentes sistemas, el mismo código Java puede ejecutarse en cualquier dispositivo con una JVM compatible.
* **Compilado e Interpretado:** El código Java se compila en bytecode que la JVM interpreta o compila justo a tiempo, lo que permite la portabilidad sin recompilar para cada plataforma.
* **Gestión Automática de Memoria:** La JVM administra la memoria y libera espacio automáticamente con un recolector de basura, evitando fugas de memoria.
* **Multihilos (Multithreading):** Java soporta la ejecución concurrente de múltiples hilos, permitiendo programas que hacen varias tareas al mismo tiempo y se comunican entre sí.

En el contexto de Java, el bytecode es el conjunto de instrucciones de bajo nivel que se genera cuando se compila un programa Java. A diferencia de otros lenguajes de programación, donde el código fuente se compila directamente a código máquina específico del sistema operativo, Java utiliza un enfoque diferente. Cuando desarrollamos en Java, el código fuente (.java) se compila utilizando el compilador de Java (javac), que traduce el código fuente a bytecode. El bytecode resultante se almacena en archivos con extensión .class. Estos archivos .class contienen instrucciones que son ejecutadas por la Máquina Virtual de Java (JVM), que es un componente crucial en la plataforma Java. El bytecode de Java es independiente de la plataforma, lo que significa que se puede ejecutar en cualquier sistema que tenga una implementación de la JVM.

Una Máquina Virtual (VM) es un entorno de ejecución que simula un sistema operativo o una computadora real. Permite ejecutar programas en un contexto controlado, independiente del hardware físico. En este entorno: Se aísla la ejecución del sistema operativo base (host) Se puede tener más de una VM por equipo físico Cada VM puede tener su propio sistema operativo, memoria, procesos, etc.

Java no compila directamente a código nativo del procesador (como lo hace C o C++), sino que compila a bytecode, un lenguaje intermedio portable. Ese bytecode es interpretado o compilado en tiempo de ejecución por la Java Virtual Machine (JVM).

**MAVEN**

Herramienta de construcción y gestión de proyectos que simplifica y automatiza tareas comunes en el desarrollo de aplicaciones Java y otros lenguajes. Su capacidad para simplificar tareas comunes y mejorar la eficiencia en el desarrollo hace que sea ampliamente utilizado en la comunidad de desarrollo y en entornos DevOps. Podemos entender a Maven desde 3 puntos de vista medianamente distintos:

**Generador de estructuras de proyecto**: ofrece la posibilidad de utilizar archetypes, que son plantillas predefinidas para crear las estructuras básicas de los proyectos. Estos arquetipos definen la organización de directorios y archivos en un proyecto, incluyendo el archivo pom.xml (Project Object Model), que es el archivo de configuración del proyecto.

Artefacto: Un producto generado por Maven, como un archivo JAR o un archivo WAR, que contiene el código compilado y las dependencias del proyecto. Es la unidad de compilación o componente indivisible dentro de la infraestructura Java y recibe un identificador que en conjunto con el GroupId y la Versión deben ser únicos. Esencialmente un artefacto está asociado al proyecto y se compone de un directorio contenedor con el archivo pom.xml y la estructura de directorios y archivos del proyecto. Para poder crear un proyecto maven a partir de un arquetipo será necesario especificar:

* GroupId: Identificador del grupo al que pertenece el proyecto. Suele ser el nombre del dominio de la organización al revés, por ejemplo, ar.edu.utnfc.backend.
* ArtifactId: Identificador único del artefacto. Es el nombre del proyecto. El nombre de artefacto normalmente se define utilizando spinal case, por ejemplo: hola-mundo.
* Version: Número de versión del artefacto. Según la convención es un conjunto de 3 números separados por punto seguidos de un guion y las palabras SNAPSHOT o RELEASE determinando si hablamos de una versión en desarrollo o de una versión definitiva del artefacto.

**Administrador de dependencias**: permite declarar las dependencias del proyecto en el archivo llamado pom.xml. Maven se encarga de descargar automáticamente las dependencias necesarias desde repositorios remotos, como Maven Central, y las incorpora en el proyecto. Una dependencia es una entidad externa (como una biblioteca, una API o un módulo) que es utilizada por una aplicación o un proyecto para cumplir ciertas funciones. Las dependencias permiten reutilizar código y funcionalidades existentes en lugar de reinventar la rueda en cada proyecto.

* Dependency Management: Declaración de dependencias en el archivo pom.xml utilizando la etiqueta . Maven descargará y gestionará estas dependencias automáticamente.
* Repositorios: Maven utiliza repositorios remotos (como Maven Central) para buscar y descargar dependencias. También puedes configurar repositorios locales o privados.
* Transitive Dependencies: Maven resuelve automáticamente las dependencias transitivas, es decir, las bibliotecas que son necesarias debido a otras dependencias.

La administración de dependencias en Maven se basa en una declaración declarativa, donde las dependencias se especifican en el archivo pom.xml y Maven se encarga de descargarlas y gestionarlas automáticamente. Además, resuelve de forma automática las dependencias transitivas, es decir, aquellas librerías que necesitan otras dependencias, evitando así problemas de compatibilidad. Maven utiliza repositorios centrales confiables, como Maven Central, para obtener las bibliotecas necesarias, y almacena las dependencias descargadas en un repositorio local (.m2), lo que evita descargas repetidas y facilita el trabajo sin conexión. Finalmente, Maven también permite, mediante la orquestación del ciclo de vida del proyecto, construir los artefactos y enviarlos al repositorio que corresponda.

**Orquestador del ciclo de vida del proyecto**: se encarga de coordinar el ciclo completo de construcción de un proyecto, desde la compilación y la ejecución de pruebas hasta la generación de artefactos binarios, como archivos JAR o WAR. A través de comandos simples, como mvn compile, mvn test o mvn package. Maven automatiza estas tareas y garantiza que el proceso de construcción sea consistente y predecible. También permite configurar y personalizar las fases de construcción según las necesidades del proyecto. Algunas fases clave del ciclo de construcción son:

* clean: Borra todos lo generado por procesos anteriores y deja el proyecto limpio para volver a comenzar
* validate: Verifica que el proyecto esté correcto y todas las dependencias estén disponibles.
* compile: Compila el código fuente del proyecto. test: Ejecuta las pruebas automatizadas.
* package: Empaqueta el código compilado en un artefacto (como un JAR o WAR).
* install: Instala el artefacto en el repositorio local para su uso en otros proyectos.
* deploy: Copia el artefacto final a un repositorio remoto para compartirlo con otros desarrolladores.

Diagrama

El contenido generado por IA puede ser incorrecto.**Pom.xl**

* src/main/java: Contiene el código fuente principal de la aplicación.
* src/main/resources: Contiene recursos (archivos de configuración, archivos de propiedades, etc.) utilizados en la aplicación principal.
* src/test/java: Contiene las pruebas unitarias y de integración.
* src/test/resources: Contiene recursos utilizados en las pruebas

El archivo pom.xml (Project Object Model) es el corazón de un proyecto de Maven. Es un archivo de texto en formato XML (Extensible Markup Language) que contiene la configuración del proyecto y a partir del cual con solo disponer del directorio de proyecto con los archivos de código y este archivo, el proyecto puede ser abierto y trabajado en cualquier IDE que admita el trabajo con proyecto Maven.

Texto

El contenido generado por IA puede ser incorrecto.<modelVersion>: Versión del modelo del proyecto de Maven (normalmente 4.0.0).

-<groupId>: Identificador del grupo al que pertenece el proyecto (usualmente el dominio de la empresa al revés).

<artifactId>: Identificador único del artefacto (nombre del proyecto).

<version>: Versión del artefacto.

<name>: Nombre descriptivo del proyecto.

<description>: Descripción del proyecto.

<dependencies>: Sección donde se declaran las dependencias del proyecto.

<properties>: Este bloque permite definir propiedades utilizadas en el proyecto. Aquí, se establece la versión de Java para la compilación y ejecución del proyecto. En este ejemplo, se utiliza Java 8.

<build>: Este bloque contiene la configuración para la construcción del proyecto.

<plugins>: Aquí se definen los plugins de construcción que se utilizarán en el ciclo de construcción del proyecto.

<plugin>: Cada plugin de construcción se define dentro de este elemento.

<groupId>: Identificador del grupo del plugin.

<artifactId>: Identificador del artefacto del plugin.

<version>: Versión del plugin.

<configuration>: Permite configurar opciones específicas del plugin. En este caso, se configura el plugin maven-compiler-plugin para que utilice la versión de Java definida en las propiedades.

Los repositorios en Maven son espacios donde se almacenan las dependencias necesarias para los proyectos. Existen tres tipos principales: el repositorio central, mantenido por la comunidad y usado por defecto para descargar librerías comunes; los repositorios remotos adicionales, que pueden configurarse para dependencias específicas o entornos restringidos; y el repositorio local, que es una carpeta (.m2) en la máquina del desarrollador donde se guardan las dependencias descargadas para evitar descargas repetidas y facilitar el trabajo sin conexión. Maven también permite crear archivos ejecutables llamados Uber JAR o Fat JAR, que incluyen el código del proyecto junto con todas sus dependencias, facilitando la portabilidad y ejecución sin necesidad de configuraciones adicionales. Finalmente, mediante el comando mvn deploy, es posible subir artefactos a repositorios remotos privados, como Nexus o Artifactory, para compartirlos entre equipos, automatizar integraciones continuas y gestionar versiones de forma centralizada.