

# ARQUITECTURA CLIENTE-SERVIDOR

- INSTRUCTOR
- I.S.C. E.T.W. M.T.W. CARLOS URIEL DE JESÚS SÁNCHEZ GONZÁLEZ
- [HTTPS://WWW.LINKEDIN.COM/IN/CARLOS-URIEL-DE-JESUS-SANCHEZ-GONZALEZ-8920A1372/](https://www.linkedin.com/in/carlos-uriel-de-jesus-sanchez-gonzalez-8920a1372/)

# ARQUITECTURA CLIENTE-SERVIDOR

- Es un modelo de comunicación en el que:
- El Cliente → solicita servicios o recursos (normalmente un programa, navegador web, aplicación móvil, etc.).
- El Servidor → responde proporcionando datos, procesando solicitudes o ejecutando servicios.
- ➡ La comunicación se realiza a través de una red (puede ser Internet o una red local).

# ARQUITECTURA CLIENTE-SERVIDOR

- El cliente envía una petición → por ejemplo, abrir una página web, consultar una base de datos, pedir un archivo.
- El servidor recibe la solicitud → la procesa (ejecuta lógica, busca información, hace cálculos).
- El servidor envía la respuesta → devuelve al cliente el resultado solicitado (ejemplo: HTML, JSON, datos de una consulta, etc.).
- El cliente muestra la información → el usuario final ve los resultados o interactúa con ellos.

# ARQUITECTURA CLIENTE-SERVIDOR



- Ejemplo Web
- Cliente: Navegador web (Chrome, Firefox, Edge).
- Servidor: Apache, Nginx o un servidor de aplicaciones (Spring Boot, Node.js).
- Flujo:
- El cliente pide: GET /index.html
- El servidor busca el archivo y lo envía.
- El cliente muestra la página en el navegador.

# ARQUITECTURA CLIENTE-SERVIDOR



- Componentes principales
- Cliente 
  - Puede ser una aplicación de escritorio, móvil o navegador web.
  - Presenta la interfaz al usuario y envía solicitudes.
- Servidor 
  - Maneja las peticiones de múltiples clientes.
  - Puede ser de distintos tipos:
    - Servidor web (páginas y APIs).
    - Servidor de base de datos (Oracle, MySQL, PostgreSQL).
    - Servidor de archivos (almacena documentos, imágenes).
- Red 
  - Medio de comunicación (Internet, LAN, etc.).
  - Utiliza protocolos como HTTP/HTTPS, TCP/IP, etc.

# ARQUITECTURA CLIENTE-SERVIDOR

-  Ventajas
-  Separación de responsabilidades → el cliente muestra, el servidor procesa.
-  Escalabilidad → varios clientes pueden conectarse al mismo servidor.
-  Seguridad → el servidor controla acceso y permisos.
-  Centralización → la lógica y datos están en un solo lugar (servidor).
-  Desventajas
-  Si el servidor falla → todos los clientes quedan sin servicio.
-  Sobrecarga → si hay demasiadas peticiones, puede saturarse.
-  Dependencia de la red → si no hay conexión, no hay comunicación.

# APACHE



- Apache HTTP Server (o simplemente Apache) es un servidor web de código abierto desarrollado por la Apache Software Foundation.
- Es uno de los servidores web más antiguos y más usados en el mundo.
- Su función principal es recibir peticiones HTTP/HTTPS de los clientes (navegadores, apps) y devolverles una respuesta, que puede ser:
  - Una página HTML
  - Un archivo (imagen, PDF, video, etc.)
  - El resultado de un programa dinámico (PHP, Java, Python, etc.)
- En resumen: Apache es el software que permite que un sitio web esté disponible en Internet.

# APACHE TOMCAT

- Apache Tomcat es un servidor de aplicaciones web de código abierto desarrollado por la Apache Software Foundation.
- Está diseñado específicamente para ejecutar aplicaciones Java que usan las especificaciones:
- Servlets 
- JSP (JavaServer Pages)
- JSTL (JSP Standard Tag Library)
- WebSockets
-  A diferencia de Apache HTTP Server (que solo sirve archivos estáticos como HTML, CSS, imágenes), Tomcat permite ejecutar aplicaciones web dinámicas escritas en Java.

# APACHE TOMCAT

- El cliente (navegador) hace una petición
  - Ejemplo: <http://localhost:8080/miapp/login>.
- Tomcat recibe la petición en su puerto
  - Por defecto, escucha en el puerto 8080.
- Busca el recurso dentro de la aplicación desplegada (WAR O JAR)
  - Si es un archivo estático (HTML, CSS, JS), lo devuelve directamente.
  - Si es un Servlet o JSP, pasa la petición al contenedor de servlets.
- El contenedor de servlets procesa la petición
  - Llama al código Java (Servlet).
  - El Servlet genera una respuesta (normalmente HTML o JSON).
- Tomcat devuelve la respuesta al cliente
  - El navegador muestra el contenido.

# MAVEN



- Maven es una herramienta de gestión y automatización de proyectos Java desarrollada por Apache.
- Se usa principalmente para:
  - Gestionar dependencias (bibliotecas externas).
  - Construir proyectos (compilar, empaquetar en JAR/WAR, etc.).
  - Estandarizar la estructura de proyectos.
  - Ejecutar fases automáticas como pruebas, empaquetado y despliegue.
- En pocas palabras: Maven te evita tener que descargar manualmente librerías y te da un flujo de trabajo estándar para desarrollar en Java.

# MAVEN



- Archivo de configuración (pom.xml) "Project Object Model"
- Define: dependencias, plugins, versiones, empaquetado, etc.
- ⚡ Maven descargará automáticamente esa librería desde el repositorio central de Maven.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ejemplo</groupId>
  <artifactId>mi-proyecto</artifactId>
  <version>1.0.0</version>

  <dependencies>
    <!-- Dependencia de Spring Boot -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>3.2.5</version>
    </dependency>
  </dependencies>
</project>
```

# MAVEN



- Repositorio
- Maven busca librerías en repositorios:
  - Local → en tu máquina (~/.m2/repository).
  - Central → en internet (Maven Central).
  - Privados → empresas pueden tener su propio repositorio.
- Ciclo de vida de construcción
- Maven tiene fases predefinidas como:
  - validate → validar configuración.
  - compile → compilar código.
  - test → ejecutar pruebas.
  - package → empaquetar en .jar o .war.
  - install → instalar en el repositorio local.
  - deploy → desplegar en un repositorio remoto.

# MAVEN



- Ventajas de Maven
- Gestión automática de dependencias.
- Estructura estándar de proyectos Java.
- Plugins para pruebas, empaquetado y despliegue.
- Integración con CI/CD (Jenkins, GitLab, GitHub Actions).
- Muy usado en frameworks como Spring, Hibernate, etc.
- Resumen
- Maven = herramienta de construcción y gestión de dependencias en proyectos Java.
- Se basa en un archivo pom.xml para definir librerías y configuraciones.
- Facilita la vida del desarrollador al estandarizar la forma de compilar, probar y empaquetar aplicaciones.

# API



- API significa Application Programming Interface (Interfaz de Programación de Aplicaciones).
- ⚡ Es un conjunto de reglas, protocolos y definiciones que permiten que dos aplicaciones se comuniquen entre sí.
- Dicho simple:
- Es como un mesero en un restaurante
- 🍔: Tú (cliente) pides algo del menú.
- El mesero (API) lleva tu pedido a la cocina (servidor).
- La cocina prepara el platillo (procesa la lógica).
- El mesero te entrega la comida (respuesta).
- Tú no hablas directo con la cocina, ni sabes cómo cocina. Solo usas el menú (documentación de la API).

# API



- 🔍 ¿Para qué sirve una API?
- Conectar sistemas distintos
  - Permite que aplicaciones, servicios o dispositivos trabajen juntos.
- Reutilizar funcionalidades
  - No necesitas reinventar la rueda: puedes usar APIs ya hechas (ej. pagos con PayPal, mapas de Google, etc.).
- Separar cliente y servidor
  - Facilita el desarrollo de apps web y móviles: el backend expone datos vía API y el frontend los consume.
- Escalabilidad y modularidad
  - Puedes actualizar el backend sin cambiar el frontend (mientras mantengas la API).
- 🔍 Tipos de APIs comunes
  - APIs Web (REST, SOAP, GraphQL) → las más usadas, funcionan sobre HTTP/HTTPS.
  - APIs de librerías → funciones ya listas dentro de un framework (ej. Java API para trabajar con fechas).
  - APIs de sistema operativo → permiten interactuar con el hardware (ej. APIs de Windows, Android).

# MVC (MODELO - VISTA - CONTROLADOR)

- Es un patrón de arquitectura de software que organiza el código de una aplicación en 3 capas separadas para mejorar la organización, mantenibilidad y escalabilidad.
-  Componentes de MVC
- Modelo (Model) 
- Representa los datos y la lógica de negocio.
- Se encarga de acceder a la base de datos, aplicar reglas de negocio, cálculos, validaciones, etc.
- Ejemplo: Clase Empleado con atributos como id, nombre, salario y métodos para calcular bonificaciones.
- Vista (View) 
- Es la interfaz de usuario (lo que el usuario ve y con lo que interactúa).
- Solo muestra la información que recibe del Modelo, pero no procesa lógica de negocio.
- Ejemplo: Una página HTML, JSP, Angular, React, etc.
- Controlador (Controller) 
- Actúa como el intermediario entre la Vista y el Modelo.
- Recibe las peticiones del usuario, llama al Modelo para procesar datos, y elige la Vista adecuada para mostrar la respuesta.
- Ejemplo: Un Servlet en Java o un controlador en Spring (@Controller).

# MVC (MODELO - VISTA - CONTROLADOR)

- Flujo típico:
  - El usuario hace una acción en la Vista (ej: clic en "Ver Empleados").
  - La Vista envía la petición al Controlador.
  - El Controlador llama al Modelo para obtener/procesar los datos.
  - El Modelo devuelve los datos al Controlador.
  - El Controlador selecciona la Vista adecuada y le pasa los datos.
  - La Vista muestra la información al usuario.
- Ventajas de MVC
  - Separación de responsabilidades → código más limpio.
  - Reutilización de código (el mismo Modelo se puede usar en varias Vistas).
  - Facilita el mantenimiento y escalabilidad.
  - Muy usado en frameworks modernos: Spring MVC, Angular (MVVM), React (similar), Laravel, Django, ASP.NET MVC.

# JDBC (JAVA DATABASE CONNECTIVITY)

- Es una API de Java que permite a las aplicaciones conectarse y ejecutar operaciones en bases de datos relacionales como Oracle, MySQL, PostgreSQL, SQL Server, etc.
  - ⚡ Piensa en JDBC como un puente entre Java y la base de datos.
- ⚡ ¿Para qué sirve JDBC?
  - Conectarse a una base de datos desde una aplicación Java.
  - Ejecutar sentencias SQL (DDL, DML, consultas).
  - Recuperar datos en forma de objetos Java (ResultSet).
  - Manejar transacciones (commit, rollback).
  - Independencia del proveedor → solo cambias el driver JDBC y puedes usar otra base de datos.
- ⚡ ¿Cómo funciona JDBC?
  - Driver JDBC → Cada base de datos tiene su propio "driver" que traduce las llamadas de JDBC al lenguaje que entiende el motor de base de datos.
    - Ejemplo: ojdbc8.jar para Oracle, mysql-connector-java.jar para MySQL.
  - Tu programa Java usa la API JDBC para conectarse.
    - Crea una conexión (Connection).
    - Ejecuta sentencias (Statement o PreparedStatement).
    - Recupera resultados (ResultSet).

# JDBC (JAVA DATABASE CONNECTIVITY)

-  **Pasos clave de JDBC**
  - Registrar el driver (en versiones modernas, basta con tener el .jar en el classpath).
  - Crear conexión con DriverManager.getConnection().
  - Crear Statement/PreparedStatement para ejecutar SQL.
  - Ejecutar consultas y obtener resultados con ResultSet.
  - Cerrar recursos (close() a ResultSet, Statement, Connection).
-  **Resumen**
  - JDBC es la API estándar de Java para interactuar con bases de datos.
  - Sirve para: conectar, consultar, actualizar y manejar datos en cualquier BD relacional.
  - Se apoya en drivers JDBC específicos para cada motor de BD.

# DAO

## DATA ACCESS OBJECT (OBJETO DE ACCESO A DATOS).

- Es un patrón de diseño que se utiliza para separar la lógica de acceso a la base de datos de la lógica de negocio de la aplicación.
- En otras palabras:
  - La capa DAO se encarga de hablar con la base de datos (consultar, insertar, actualizar, borrar).
  - La capa de negocio solo consume métodos del DAO sin preocuparse de cómo se conecta o qué SQL se usa.
- ¿Para qué sirve DAO?
- Separar responsabilidades
  - El acceso a la BD está en un lugar, y la lógica de negocio en otro.
- Facilitar mantenimiento
  - Si cambias la base de datos (ej. de Oracle a MySQL), solo actualizas el DAO.
- Reutilización de código
  - Puedes usar el mismo DAO en distintas partes de la app.
- Mayor seguridad y limpieza
  - Se evita mezclar consultas SQL con la lógica de negocio.

# HIBERNATE

- Hibernate es un framework de mapeo objeto-relacional (ORM) para Java.
- ORM (Object Relational Mapping) significa que mapea (traduce) objetos Java a tablas de una base de datos relacional y viceversa.
- Con Hibernate:
  - No tienes que escribir tanto SQL manualmente.
  - Puedes trabajar con objetos Java y Hibernate se encarga de transformarlos en sentencias SQL que entiende la base de datos.
- ¿Para qué sirve Hibernate?
  - Simplificar el acceso a la base de datos
    - En lugar de usar JDBC con SQL, trabajas con objetos (Empleado, Producto, etc.).
  - Evitar código repetitivo
    - Hibernate genera automáticamente las consultas SQL.
  - Independencia de la base de datos
    - Cambiar de MySQL a Oracle o PostgreSQL es más fácil (solo cambias la configuración del driver).
  - Gestión avanzada
    - Maneja transacciones, caché, lazy loading (carga diferida), relaciones (@OneToMany, @ManyToMany, etc.).

# HIBERNATE



- Cómo funciona Hibernate?
- Defines una clase Java (modelo) y la anotas con JPA/Hibernate annotations (@Entity, @Table, etc.).
- Hibernate mapea esa clase a una tabla de la base de datos.
- Usas métodos de Hibernate (save, update, delete, query) en lugar de SQL.
- Hibernate traduce esas operaciones a sentencias SQL y las ejecuta en la base de datos.

# HIBERNATE

- Ventajas de Hibernate

- Reduce código repetitivo de JDBC.
- Compatible con múltiples bases de datos.
- Maneja relaciones complejas entre tablas con anotaciones.
- Soporta HQL (Hibernate Query Language) → consultas orientadas a objetos.
- Se integra fácilmente con Spring Framework.

- Resumen

- Hibernate = framework ORM para Java.
- Sirve para: trabajar con bases de datos usando objetos en lugar de SQL puro.
- Facilita el desarrollo, reduce errores y hace más mantenible la aplicación.

# JPA (JAVA PERSISTENCE API)

- Es una especificación oficial de Java para el mapeo objeto-relacional (ORM).
- Define cómo las aplicaciones Java deben interactuar con bases de datos usando objetos, sin preocuparse por escribir SQL puro todo el tiempo.
- Hibernate, EclipseLink y OpenJPA son implementaciones de JPA.
- ➡ En resumen: JPA es la “regla” y Hibernate es una de sus implementaciones más populares.
- ⚙ ¿Para qué sirve JPA?
  - Persistir objetos en la base de datos
    - Guardar, actualizar o eliminar objetos Java como registros en tablas.
  - Mapeo objeto-relacional
    - Permite que las clases y atributos Java se correspondan con tablas y columnas de la base de datos.
  - Consultar la base de datos usando objetos
    - Con JPQL (Java Persistence Query Language) puedes hacer consultas orientadas a objetos.
  - Independencia del proveedor
    - Si cambias de Hibernate a EclipseLink, tu código JPA sigue funcionando.
  - Manejo de relaciones
    - Maneja asociaciones como @OneToMany, @ManyToOne, @ManyToMany entre entidades.

# JPA (JAVA PERSISTENCE API)

- Ventajas de JPA
- Abstracción de la base de datos → menos SQL manual.
- Código más limpio y mantenible.
- Compatible con múltiples proveedores (Hibernate, EclipseLink...).
- Manejo de relaciones y transacciones de forma nativa.
- Resumen rápido:
  - JPA = especificación para persistir objetos Java en bases de datos relacionales.
  - Sirve para: almacenar, actualizar, eliminar y consultar objetos Java sin escribir SQL directamente.
  - Hibernate es la implementación más usada de JPA.

# JPA (JAVA PERSISTENCE API)

Característica

JPA

Hibernate

Tipo

Especificación / API

Implementación concreta de ORM

Propósito

Definir reglas para persistencia  
en Java

Proveer la funcionalidad  
completa de ORM siguiendo JPA

Flexibilidad

Puede usar distintos proveedores

Solo Hibernate (aunque cumple  
JPA)

Características extra

Ninguna, solo la especificación

Caché, HQL, Lazy loading  
avanzado, etc.