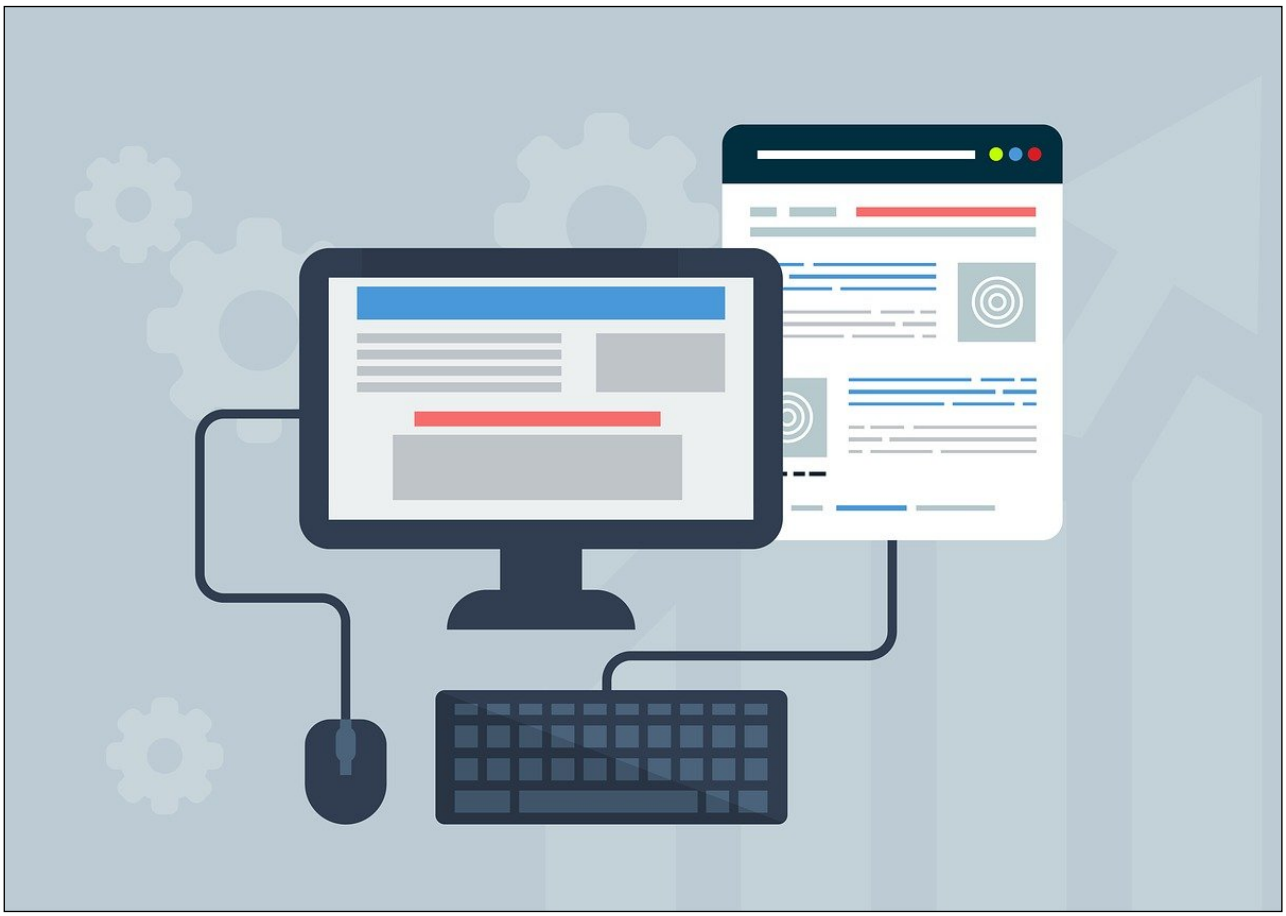
Unidad 1. Implantación de arquitecturas web.

Módulo: Despliegue de aplicaciones web (2ºDAW)

Profesora: Isabel Soriano



*El **resultado de aprendizaje** asociado a esta unidad es el siguiente:*

RA1. Implanta arquitecturas Web analizando y aplicando criterios de funcionalidad.

ÍNDICE

1. ¿Qué es una arquitectura web?

Una arquitectura web es la estructura que define cómo se organizan, comunican y funcionan los diferentes componentes de una aplicación o sitio web. Es decir, es el diseño técnico que determina cómo se construye y opera un sistema web.

1.1 Componentes principales

a. Cliente (Frontend)

- Envía solicitudes al servidor y muestra las respuestas.
- Es la parte visible para el usuario: la interfaz gráfica.
- Suele ejecutarse en un navegador web.
- Utiliza tecnologías como HTML, CSS y JavaScript.
- Puede incluir frameworks como React, Angular, Vue.js.

b. Servidor (Backend)

- Procesa las solicitudes del cliente.
- Se encarga de la lógica de negocio, la autenticación, y la conexión con bases de datos.
- Usa tecnologías como Node.js, Python, Java, PHP, Ruby, etc.

c. Base de datos

- Almacena y gestiona la información que utiliza la aplicación.
- Tipos: relacionales (como MySQL, PostgreSQL), NoSQL (como MongoDB, Redis).

d. Protocolo de comunicación

- Normalmente HTTP o HTTPS.
- Se usa para enviar solicitudes y recibir respuestas entre cliente y servidor.

2. Modelos de arquitectura web

Las aplicaciones web pueden ser construidas utilizando diferentes arquitecturas, cada una con sus propios pros y contras. Elegir la arquitectura adecuada es fundamental para satisfacer las necesidades específicas de la aplicación y la organización.

2.1 Arquitectura Monolítica

En una arquitectura monolítica, todo el código de la aplicación se encuentra en un único proyecto o servidor. Esta arquitectura es más sencilla de implementar pero puede ser difícil de escalar y mantener a medida que la aplicación crece.

Ventajas:

- Facilidad de desarrollo y despliegue inicial.
- Simplifica el manejo de transacciones y sesiones.
- Menos sobrecarga en la comunicación entre componentes.

Desventajas:

- Dificultad para escalar.
- Mayor riesgo de fallos en toda la aplicación.
- Ciclos de despliegue más largos.
- Menor flexibilidad en la adopción de nuevas tecnologías.

EJEMPLO: Un **sistema de gestión hospitalaria** construido como monolítico:

- Todo está en una sola aplicación: la interfaz web, la lógica de pacientes, citas, facturación y el acceso a la base de datos.
- Si hay que añadir un nuevo módulo (por ejemplo, un sistema de notificaciones por SMS), se debe modificar y desplegar la aplicación completa.

2.2 Arquitectura de capas

Esta arquitectura divide la aplicación en diferentes capas lógicas, donde cada capa tiene una responsabilidad específica. Las capas típicas incluyen la capa de presentación, la capa de lógica de negocio y la capa de acceso a datos. Cada capa se comunica con la capa adyacente a través de interfaces bien definidas.

Ventajas:

- Mejora la modularidad y la reutilización del código.
- Permite cambios en una capa sin afectar a las demás.
- Facilita la escalabilidad y el mantenimiento del sistema.

Desventajas:

- Menor flexibilidad en la adopción de nuevas tecnologías.
- Menor mantenimiento del sistema.
- Menor escalabilidad.
- Menor flexibilidad para el cambio de tecnologías.

EJEMPLO: Un **e-commerce (como Amazon)** usa arquitectura en capas:

- **Capa de presentación:** el frontend web o app móvil que ve el usuario (React, Angular, etc.).
- **Capa de lógica de negocio:** las reglas de compra, validación de stock, cálculo de envío y descuentos (implementadas en Node.js, Java, .NET, etc.).
- **Capa de acceso a datos:** consultas a la base de datos (MySQL, PostgreSQL, MongoDB) donde se almacenan productos, usuarios y pedidos.
-

Así, si Amazon quiere cambiar la interfaz web, no necesita tocar la lógica de negocio ni la base de datos; las capas permiten trabajar de manera independiente.

2.3 Arquitectura de servicios web

Esta arquitectura se basa en la comunicación entre diferentes servicios a través de protocolos web estándar, como HTTP. Cada servicio es una unidad independiente que se puede desarrollar, desplegar y escalar de forma independiente. Los servicios se comunican entre sí para cumplir con los requisitos de la aplicación.

Ventajas:

- Favorece la modularidad y la independencia de los servicios.
- Permite la integración de diferentes tecnologías y lenguajes de programación.
- Facilita la escalabilidad horizontal, ya que los servicios se pueden replicar y distribuir en múltiples servidores.

Desventajas:

- Mantenimiento más costoso: monitorear y administrar muchos servicios requiere infraestructura adicional.
- Dependencia de red: si hay problemas de conectividad, los servicios dejan de funcionar.
- Complejidad: diseñar e integrar múltiples servicios requiere buena planificación y experiencia.

EJEMPLO: Un **banco digital** puede usar arquitectura de servicios web:

- Servicio de autenticación: para validar usuarios y contraseñas.
- Servicio de cuentas: para consultar saldos y movimientos.
- Servicio de pagos: para transferencias y pagos de servicios.
- Servicio de notificaciones: para enviar correos o SMS.

Cada servicio es independiente y puede ser consumido por la aplicación web, la app móvil o incluso por aplicaciones de terceros (ej. pago de facturas desde otra plataforma).

2.4 Arquitectura de Microservicios

La arquitectura de microservicios divide la aplicación en múltiples servicios pequeños e independientes que interactúan entre sí. Cada microservicio es responsable de una funcionalidad específica y puede ser desarrollado, desplegado y escalado de manera independiente.

Ventajas:

- Escalabilidad independiente.
- Facilita el desarrollo y la implementación por equipos separados.
- Mayor resiliencia, ya que el fallo de un servicio no afecta a toda la aplicación.
- Flexibilidad para usar diferentes tecnologías y lenguajes en diferentes microservicios.

Desventajas:

- Mayor complejidad en la gestión y el despliegue.
- Requiere una configuración de infraestructura avanzada.
- Aumenta la sobrecarga de comunicación entre servicios.

EJEMPLO: Netflix:

- Tiene microservicios independientes para la gestión de usuarios, catálogos de películas, recomendaciones personalizadas, streaming de video, facturación, etc.
- Cada microservicio se despliega y escala según su necesidad.
- Si el servicio de recomendaciones falla, la plataforma sigue funcionando para reproducir contenido.

Otros ejemplos: Amazon, Uber, Spotify también migraron a microservicios para soportar millones de usuarios y crecer más rápido.

2.5 Arquitectura Serverless

En una arquitectura serverless, las operaciones del servidor están completamente gestionadas por un proveedor de nube. Los desarrolladores no tienen que preocuparse por la infraestructura, sino solo por el código de la aplicación.

Ventajas:

- Escalabilidad automática.
- Pago por uso: solo se cobran los recursos utilizados.
- Alto nivel de abstracción en la gestión de infraestructura.
- Reducción de la complejidad en el despliegue y la administración del servidor.

Desventajas:

- Dependencia del proveedor de servicios.
- Limitaciones en el tiempo de ejecución y recursos.
- Latencia de arranque en frío.
- Desafíos en el monitoreo y la depuración.

EJEMPLO: Un **chatbot** de atención al cliente:

- Cada vez que un usuario envía un mensaje, se activa una función serverless.
- Esa función procesa el mensaje (usando IA o reglas) y responde.
- Si no hay usuarios activos, no se consume nada y no se paga por servidores ociosos.

3. Tipos de despliegue de aplicaciones web

El despliegue de aplicaciones web puede variar según el contexto de la aplicación y la infraestructura disponible. Aquí se presentan varios tipos:

3.1 Despliegue de Páginas

El despliegue de páginas web estáticas se refiere a la publicación de contenido que no cambia dinámicamente. Este tipo de despliegue es más sencillo y se utiliza para sitios web informativos, blogs y páginas de destino.

- **Tecnologías comunes:** HTML, CSS, JavaScript
- **Hosting:** Proveedores como GitHub Pages, Netlify, y Vercel.
- **Desarrollo de la Página Web:** Se crean los archivos HTML, CSS y JavaScript necesarios.
- **Subida de Archivos al Servidor de Hosting:** Estos archivos se suben a un servidor de hosting mediante FTP, Git o una plataforma de despliegue.
- **Configuración del Dominio:** Se configura un dominio para hacer que la página sea accesible públicamente.
- **Configuración Opcional de CDN y SSL:** Para mejorar el rendimiento y la seguridad, se puede configurar una red de distribución de contenido (CDN) y un certificado SSL.
- **Documentación:** Es fundamental documentar cada paso del proceso para asegurar la repetibilidad y facilitar la resolución de problemas futuros.
-

3.2 Despliegue en Contenedores

El despliegue en entornos de contenedores o nube es más complejo y se utiliza para aplicaciones dinámicas que requieren escalabilidad y flexibilidad. Tecnologías comunes: Docker, Kubernetes.

Ventajas:

- Portabilidad.
- Consistencia entre entornos de desarrollo y producción.
- Mejor uso de recursos.

Desventajas:

- Seguridad: una vulnerabilidad en el contenedor puede afectar al host.
- Sobrecarga de gestión en producción: en sistemas grandes se requieren herramientas de orquestación (Kubernetes, OpenShift), lo que aumenta la complejidad operativa.
- Compatibilidad parcial: no todas las aplicaciones tradicionales funcionan bien en contenedores

3.3 Despliegue en la Nube

Algunos proveedores comunes son: AWS, Google Cloud, Azure.

Ventajas:

- Escalabilidad automática.
- Servicios gestionados (bases de datos, colas, almacenamiento).
- Alta disponibilidad.

Desventajas:

- Dependencia del proveedor.
- Privacidad y seguridad de datos: los datos residen en servidores externos; si no hay medidas adecuadas, pueden exponerse a accesos no autorizados.
- Dependencia de Internet

4. Estructura y recursos que componen una aplicación web.

Una **aplicación web** está compuesta por diversos **recursos**, que son todos los elementos necesarios para que funcione correctamente, tanto en el lado del cliente (navegador) como en el servidor.

4.1 Recursos del lado del cliente (Frontend)

Estos se ejecutan en el navegador del usuario y son responsables de mostrar la interfaz y permitir la interacción.

- **HTML (HyperText Markup Language):**
  Estructura básica del contenido web (títulos, párrafos, botones, formularios, etc.).
- **CSS (Cascading Style Sheets):**
  Estilo y diseño visual (colores, fuentes, márgenes, animaciones, etc.).
- **JavaScript:**
  Permite interactividad (validaciones, animaciones, peticiones al servidor, etc.).
- **Imágenes y multimedia:**
  Logos, íconos, banners, videos, etc.
- **Fuentes y archivos estáticos:**
  Tipografías personalizadas, archivos PDF, SVG, etc.

4.2. Recursos del lado del servidor (Backend)

Son los que procesan la lógica de la aplicación, gestionan datos y se comunican con la base de datos.

- **Lenguajes de programación del servidor:**
  Como PHP, Python, Java, JavaScript (Node.js), Ruby, etc.
- **Frameworks web:**
  Herramientas que simplifican el desarrollo del backend, como Django, Laravel, Express.js, etc.

- **Archivos de configuración:**
  Definen variables del entorno, rutas, puertos, seguridad, etc. (como .env, config.json, etc.)

- **Módulos o librerías del servidor:**
  Paquetes reutilizables que ayudan a agregar funcionalidades (como autenticación, validación, envío de correos, etc.).

4.3. Base de datos

Recurso esencial para almacenar, modificar y consultar datos de usuarios, productos, publicaciones, etc.

- **Bases de datos relacionales:** MySQL, PostgreSQL, SQL Server

- **Bases de datos no relacionales:** MongoDB, Firebase, Redis

4.4. Recursos de red y comunicación

- **APIs (Interfaces de programación):**
  Permiten la comunicación entre el frontend y el backend, o con otros servicios.

- **Protocolo HTTP/HTTPS:**
  Para enviar y recibir solicitudes/respuestas entre cliente y servidor.

- **WebSockets:**
  Para comunicación en tiempo real (como chats o notificaciones en vivo).

4.5. Recursos de seguridad

- **Certificados SSL/TLS:**
  Encriptan la comunicación (HTTPS).

- **Autenticación y autorización:**
  Sistemas para controlar el acceso (tokens, sesiones, OAuth, etc.).

4. 6. Recursos de despliegue y alojamiento

- **Servidores web:** Apache, Nginx, IIS
- **Sistemas operativos del servidor:** Linux (Ubuntu, CentOS), Windows Server
- **Servicios en la nube:** AWS, Azure, Heroku, Vercel, Netlify
- **Contenedores:** Docker, Kubernetes

4.7 Ejemplo de estructura de una aplicación web

A continuación, se muestra un ejemplo de la estructura de una aplicación web, que usa:

- **Backend:** PHP ejecutado en **servidor Apache**
- **Base de datos:** MySQL
- **Frontend:** HTML, CSS y JavaScript (sin frameworks)

Estructura de proyecto con Apache + PHP + MySQL:

```
/mi-aplicacion-web
├── /public_html/            # Carpeta raíz del servidor (accesible desde el navegador)
│   ├── /assets/             # Imágenes, CSS, JS
│   │   ├── /css/            # Hojas de estilo
│   │   │   └── styles.css
│   │   ├── /js/             # Scripts del frontend
│   │   │   └── app.js
│   │   └── /img/            # Archivos de imagen
│   ├── /includes/           # Archivos reutilizables (conexiones, cabeceras, etc.)
│   │   ├── db.php           # Conexión a la base de datos MySQL
│   │   └── header.php       # Cabecera común de las páginas
│   ├── /pages/              # Páginas del sitio
│   │   ├── login.php
│   │   ├── register.php
│   │   └── dashboard.php
│   ├── index.php            # Página principal
│   └── .htaccess            # Configuración específica para Apache
├── /sql/                    # Script para crear base de datos y tablas
│   └── init.sql
├── README.md                # Documentación del proyecto
└── .gitignore               # Ignora archivos del control de versiones
```

Recurso / Archivo	Función
`index.php`	Página principal de la app, puede ser el login o la portada
`/includes/db.php`	Script para conectar con la base de datos MySQL
`/assets/css/styles.css`	Diseño visual de la app
`/pages/dashboard.php`	Página privada tras login (requiere autenticación)
`.htaccess`	Permite configurar redirecciones, URLs limpias, seguridad, etc.
`/sql/init.sql`	Script SQL para crear las tablas (usuarios, productos, etc.)

5. Descriptor de despliegue.

Un **descriptor de despliegue** es un **archivo de configuración** que le dice al servidor **cómo debe ejecutarse una aplicación web**, qué rutas manejar, qué recursos proteger, cómo iniciar componentes, entre otras cosas.

En este módulo, vamos a instalar, configurar y administrar tanto servidores web (como Apache) como servidores de aplicaciones (como Tomcat). El concepto de descriptor de despliegue está directamente relacionado con los servidores de aplicaciones, donde define cómo se comporta la aplicación dentro del servidor. Sin embargo, en los servidores web también existen archivos de configuración similares (como .htaccess en Apache), que cumplen funciones relacionadas con el despliegue y el comportamiento del sitio.

Por otro lado, cabe destacar que en muchas arquitecturas web es común usar conjuntamente servidores web y servidores de aplicaciones. Por ejemplo, el servidor web puede recibir todas las solicitudes entrantes y redirigir aquellas que requieren procesamiento dinámico al servidor de aplicaciones, que se encarga de generar la respuesta adecuada.

5.1 Servidores web

Un **servidor web** es un software que:

- **Recibe solicitudes HTTP o HTTPS** desde un navegador.
- **Entrega archivos estáticos** como HTML, CSS, JavaScript, imágenes, etc.
- Puede ejecutar código **del lado del servidor en lenguajes como PHP** (si tiene módulos instalados).

Algunos ejemplos son: Apache HTTP Server, Nginx y LiteSpeed.

En el caso de servidores web, como puede ser un **entorno LAMP** (Linux, Apache, MySQL, PHP), no existe un "descriptor de despliegue" formal como en Java, pero sí hay archivos de configuración equivalentes que cumplen funciones similares para controlar el comportamiento del servidor web y la aplicación.

El archivo .htaccess se considera el **descriptor de despliegue más cercano** en Apache, ya que permite configurar:

- URLs amigables (reescritura de rutas)
- Redirecciones
- Seguridad (control de acceso)
- Páginas de error personalizadas
- Configuraciones PHP por directorio
- Comportamiento del caché

La **estructura del proyecto** podría ser la siguiente, donde se incluye el descriptor de despliegue:

```
/mi-aplicacion-lamp/
├── /public_html/           ← Carpeta raíz del sitio (expuesta al navegador)
│   ├── index.php           ← Entrada principal de la app
│   ├── login.php           ← Página de login
│   ├── /css/
│   ├── /js/
│   ├── /img/
│   ├── /private/           ← Carpeta protegida (por .htaccess)
│   └── .htaccess           ← Descriptor de despliegue (Apache)
├── /sql/
│   └── init.sql            ← Script para crear base de datos MySQL
└── README.md
```

Ejemplo de archivo .**htaccess:**

```
# Activar la reescritura de URLs
RewriteEngine On

# Redirigir todas las rutas a index.php si no existen como archivos o carpetas
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.+)$ index.php?url=$1 [QSA,L]

# Bloquear acceso a la carpeta /private
<Directory "/var/www/html/private">
    Order allow,deny
    Deny from all
</Directory>

# Página de error personalizada
ErrorDocument 404 /404.php

# Establecer configuración PHP personalizada
php_value upload_max_filesize 10M
php_value memory_limit 128M
```

Sección	Función
`RewriteEngine On`	Activa el módulo de reescritura de URLs
`RewriteRule`	Permite URLs limpias como `/productos/123` en lugar de `?id=123`
`<Directory>`	Bloquea el acceso a una carpeta sensible
`ErrorDocument`	Define una página personalizada para errores HTTP
`php_value`	Cambia valores de configuración de PHP para este directorio

En LAMP no hay un descriptor estándar centralizado como web.xml, pero:

- **.htaccess** actúa como descriptor a nivel de Apache.
- Archivos PHP como **index.php** pueden actuar como enrutadores de la lógica.

Puedes tener múltiples **.htaccess** si tienes subdirectorios, lo que permite despliegues modulares por carpeta.

5.2 Servidores de aplicaciones

Un **servidor de aplicaciones** es un software más avanzado que:

- Ejecuta **lógica de negocio compleja** y aplicaciones completas del lado del servidor.
- Soporta tecnologías como **Java EE, Servlets, JSP, EJB, Spring**, etc.
- Maneja conexiones con bases de datos, seguridad, transacciones, y más.

Algunos ejemplos son:Apache Tomcat, WildFly (ex-JBoss), GlassFish, WebLogic e IBM WebSphere.

En **aplicaciones Java** (por ejemplo, con Servlets o JSP), el descriptor de despliegue es el archivo **web.xml**.

EJEMPLO: Supongamos que tienes un Servlet llamado **HolaServlet.java** que responde a **/hola**.

La **estructura del proyecto** podría ser la siguiente, donde se incluye el descriptor de despliegue:

```
/mi-aplicacion.war
├── /WEB-INF/
│   ├── web.xml            ← Descriptor de despliegue
│   └── /classes/
│       └── HolaServlet.class
└── index.jsp
```

- **mi-aplicacion.war** → el paquete completo de la aplicación. Se despliega directamente en el servidor de aplicaciones. Dentro lleva código, configuraciones y recursos.
- **/WEB-INF/** → Carpeta especial no accesible desde el navegador (es privada). Contiene configuraciones y clases necesarias para que la aplicación funcione. Todo lo que esté aquí es solo para el servidor, no para el usuario final.
- **web.xml** → configuración (descriptor de despliegue). Define cómo se comporta la aplicación.
- **/classes/** → clases Java compiladas (.class).
- **HolaServlet.class** → un servlet de ejemplo. Procesa peticiones HTTP y devuelve respuestas dinámicas.
- **index.jsp** → Página JSP (*Java Server Pages, tecnología que permite crear páginas web dinámicas mezclando código Java con HTML, CSS y JavaScrip*t.) accesible directamente desde el navegador. Es parte de la capa de presentación (interfaz con el usuario). El servidor traduce este JSP a un servlet y genera HTML dinámico.

Y el contenido del **descriptor 'web.xml'** sería el siguiente:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                             http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

    <!-- Definición del servlet -->
    <servlet>
        <servlet-name>HolaServlet</servlet-name>
        <servlet-class>HolaServlet</servlet-class>
    </servlet>

    <!-- Mapeo del servlet a una URL -->
    <servlet-mapping>
        <servlet-name>HolaServlet</servlet-name>
        <url-pattern>/hola</url-pattern>
    </servlet-mapping>

    <!-- Página por defecto si no se especifica ruta -->
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

Elemento	Función
`<servlet>`	Declara el servlet y su clase Java asociada
`<servlet-mapping>`	Define la URL en la que estará disponible ese servlet (`/hola`)
`<welcome-file-list>`	Indica la página predeterminada cuando se accede al directorio raíz

Se debe tener en cuenta que el descriptor de despliegue, no es obligatorio en versiones recientes de Java. Se pueden usar simplemente anotaciones en el código Java (como @WebServlet, @WebFilter, etc.) No obstante, es recomendable el uso del descriptor web.xml. el cual debe tener ese nombre exacto y estar en la carpeta /WEB-IN

6. Servidor web Apache.

Apache es un servidor web de código abierto que ha sido fundamental en el crecimiento de Internet, ya que permite a cualquier persona instalarlo, configurarlo y modificarlo de acuerdo con sus necesidades sin costo de licencias.

Además de Apache, existen otros proyectos de Software Libre muy utilizados para ofrecer servicios web, entre los que destacan:

- Nginx: Servidor web ligero y muy eficiente, ampliamente usado para sitios de alto tráfico.
- Lighttpd: Optimizado para entornos con recursos limitados.
- Tomcat: Servidor de aplicaciones Java desarrollado bajo el paraguas de la Fundación Apache.
- Caddy: Servidor web moderno que incluye HTTPS automático.

6.1 Comandos básicos

A continuación, se muestra una tabla con los comandos básicos para instalar y configurar un servidor Apache.

Comando	Función
`sudo apt update`	Actualiza la lista de paquetes disponibles.
`sudo apt install apache2 -y`	Instala el servidor Apache.
`sudo systemctl start apache2`	Inicia el servicio de Apache.
`sudo systemctl enable apache2`	Habilita Apache para que arranque automáticamente al iniciar el sistema.
`sudo systemctl status apache2`	Muestra el estado actual del servicio Apache.
`sudo ufw allow 'Apache'`	Abre el firewall para permitir el tráfico HTTP y HTTPS.
`sudo nano /var/www/html/index.html`	Edita la página principal del sitio web.
`sudo systemctl restart apache2`	Reinicia Apache para aplicar cambios de configuración.
`apache2 -v`	Verifica la versión instalada de Apache.

6.2 Ficheros de configuración

En la siguiente tabla, se muestran los archivos de configuración más importantes, para instalar y configurar un servidor Apache

Nombre del fichero	Ruta	Función
`apache2.conf`	`/etc/apache2/apache2.conf`	Archivo principal de configuración de Apache.
`ports.conf`	`/etc/apache2/ports.conf`	Define los puertos en los que escucha Apache (80, 443, etc.).
`000-default.conf`	`/etc/apache2/sites-available/000-default.conf`	Configuración del sitio web por defecto (VirtualHost).
`default-ssl.conf`	`/etc/apache2/sites-available/default-ssl.conf`	Configuración del sitio por defecto con soporte SSL/TLS.
`.htaccess`	Dentro de `/var/www/html/` o del directorio del sitio	Permite aplicar configuraciones locales (redirecciones, permisos, reescrituras).
`envvars`	`/etc/apache2/envvars`	Define variables de entorno usadas por Apache (usuario, grupo, etc.).
Archivos en `sites-available/`	`/etc/apache2/sites-available/`	Contienen configuraciones de distintos sitios virtuales.
Archivos en `sites-enabled/`	`/etc/apache2/sites-enabled/`	Enlazan los sitios activados que Apache cargará al iniciar.
Archivos en `mods-available/`	`/etc/apache2/mods-available/`	Configuración de módulos disponibles (rewrite, ssl, etc.).
Archivos en `mods-enabled/`	`/etc/apache2/mods-enabled/`	Enlazan los módulos activados que se cargan en Apache.

6.3 Directivas del fichero 'apache2.conf'

El fichero apache2.conf es el archivo principal de configuración del servidor web Apache en sistemas basados en Debian/Ubuntu.

Su función es definir las directrices globales que determinan el comportamiento general del servidor, tales como:

- Los directorios donde se encuentran los sitios web y los módulos.
- Las políticas de seguridad (permisos, usuarios y grupos con los que se ejecuta el servicio).

- La carga de otros archivos de configuración (por ejemplo, incluye referencias a ports.conf, sites-enabled/, mods-enabled/).
- Parámetros de rendimiento como el número de procesos hijos, el uso de memoria o los límites de conexiones.

La relevancia del fichero apache2.conf radica en que cualquier cambio en este archivo afecta a todo el servidor, no solo a un sitio en particular. A diferencia de los ficheros dentro de sites-available o .htaccess que afectan a un sitio web o directorio específico, las directivas en apache2.conf se aplican de forma global.

En la siguiente tabla, se muestran las directivas más comunes de apache2.conf para un servidor Apache básico en Ubuntu/Debian. Los ejemplos están pensados para un servidor que aloja un sitio en /var/www/html y escucha en el puerto 80:

Directiva	Función	Ejemplo
`ServerRoot`	Define el directorio base donde se encuentran los archivos de configuración y binarios de Apache.	`ServerRoot "/etc/apache2"`
`Listen`	Especifica el puerto en el que Apache escuchará las peticiones.	`Listen 80`
`User`	Usuario del sistema con el que se ejecuta Apache (seguridad).	`User www-data`
`Group`	Grupo del sistema con el que se ejecuta Apache.	`Group www-data`
`ServerAdmin`	Dirección de correo del administrador del servidor.	`ServerAdmin admin@midominio.com`
`DocumentRoot`	Define el directorio raíz donde se almacenan los archivos del sitio web.	`DocumentRoot /var/www/html`
`<Directory>`	Bloque de configuración que controla permisos y opciones en un directorio específico.	`<Directory /var/www/html>\n Options Indexes FollowSymLinks\n AllowOverride All\n Require all granted\n</Directory>`
`ErrorLog`	Archivo donde se registran los errores del servidor.	`ErrorLog ${APACHE_LOG_DIR}/error.log`
`CustomLog`	Define el archivo y el formato del registro de accesos.	`CustomLog ${APACHE_LOG_DIR}/access.log combined`

Con estas directivas, el servidor Apache quedaría listo para servir páginas estáticas desde /var/www/html, con logs activados, permisos adecuados y un administrador definido.