Aplicaciones Web. Modelo de capas

La arquitectura de capas en aplicaciones web es un patrón de diseño estructural que organiza el software en diferentes niveles o capas, cada una con una responsabilidad específica. Este enfoque facilita la separación de responsabilidades, mejora la mantenibilidad, la escalabilidad y promueve el desarrollo modular.

Capas comunes en una arquitectura web

1. Capa de Presentación (Front-End)

- Responsabilidad: Se encarga de la interacción con el usuario. Aquí se gestiona todo lo relacionado con la interfaz gráfica (UI), la lógica de presentación y la experiencia de usuario (UX).
- Tecnologías: HTML, CSS, JavaScript, frameworks como React, Angular o Vue.js.
- Ejemplo: Formularios, botones, menús, diseño de la página web.

2. Capa de Lógica de Negocio (Back-End o Capa de Aplicación)

- Responsabilidad: Implementa la lógica central del negocio, es decir, las reglas y el procesamiento que sigue la aplicación. Esta capa actúa como el puente entre la presentación y los datos.
- Tecnologías: Java, Python, PHP, Ruby, Node.js, .NET.

- Ejemplo: Validación de datos, gestión de usuarios, procesamiento de pagos.

3. Capa de Acceso a Datos

- Responsabilidad: Gestiona el acceso a las fuentes de datos, como bases de datos, APIs externas o archivos. Aquí se encapsulan las operaciones de lectura/escritura de datos.
- Tecnologías: SQL (MySQL, PostgreSQL), NoSQL (MongoDB), ORMs (Hibernate, Sequelize), servicios web, API REST.
- Ejemplo: Consultas a la base de datos, inserción y modificación de registros.

4. Capa de Datos (Base de Datos)

- Responsabilidad: Es donde se almacenan los datos de la aplicación, ya sean datos estructurados en bases de datos relacionales o no relacionales.
- Tecnologías: MySQL, MongoDB, SQLite, Oracle, etc.
- Ejemplo: Almacén de datos de clientes, productos, transacciones.

Beneficios de la arquitectura en capas

- Modularidad: Cada capa se desarrolla y prueba de forma independiente, lo que facilita el mantenimiento y la evolución del software.
- Reutilización: Las capas se pueden reutilizar en diferentes partes de la aplicación o incluso en otras aplicaciones.

- Facilidad de mantenimiento: Al estar organizadas por responsabilidades, los cambios en una capa afectan mínimamente a las demás.
- Escalabilidad: Es posible escalar cada capa de manera independiente, optimizando el uso de recursos en función de las necesidades de la aplicación.
- Seguridad: La separación de capas permite controlar el acceso a los datos y a la lógica de negocio desde la interfaz, lo que mejora la seguridad.

Ejemplo de flujo en una aplicación de capas

1. El usuario interactúa con el front-end (Capa de Presentación) para enviar un formulario de registro.
2. Los datos son validados y procesados en la Capa de Lógica de Negocio.
3. La Capa de Acceso a Datos se conecta a la Base de Datos para almacenar los datos del nuevo usuario.
4. Finalmente, se muestra un mensaje de éxito al usuario en la Capa de Presentación.

Este modelo es común en aplicaciones web modernas y fomenta un desarrollo más limpio y organizado.

Patrón MVC

El patrón MVC (Model-View-Controller) es un patrón de diseño arquitectónico muy utilizado en el desarrollo de aplicaciones web. Su principal objetivo es **separar la lógica de negocio, la interfaz de usuario y el control de flujo**, lo que facilita el mantenimiento, la escalabilidad y la organización del código. A continuación, se describen sus tres componentes principales:

1. Model (Modelo)

- Responsabilidad: Es la capa que gestiona los datos y la lógica de negocio de la aplicación. El modelo interactúa con la base de datos, realiza cálculos, valida reglas de negocio y envía los datos a la vista cuando es necesario.
- Ejemplo: En una aplicación de tienda online, el modelo se encargaría de gestionar los productos, los usuarios, las órdenes de compra, etc.
- Tecnologías: En aplicaciones web, el modelo generalmente está asociado con ORM (Object-Relational Mapping) como Eloquent (Laravel), ActiveRecord (Ruby on Rails) o frameworks que permiten la interacción con bases de datos como Hibernate (Java).

2. View (Vista)

- Responsabilidad: La vista es la capa encargada de mostrar la interfaz gráfica y los datos al usuario. No contiene lógica de

negocio, solo se encarga de presentar la información obtenida del modelo.

- Ejemplo: En la misma tienda online, la vista podría ser el HTML que muestra la lista de productos al usuario con su precio y descripción.
- Tecnologías: HTML, CSS, JavaScript, junto con frameworks o motores de plantillas como Blade (Laravel), JSP (Java), EJS (Node.js), etc.

3. Controller (Controlador)

- Responsabilidad: El controlador actúa como intermediario entre el modelo y la vista. Recibe las solicitudes del usuario (generalmente a través de la URL), procesa esas solicitudes utilizando los datos del modelo y selecciona la vista adecuada para mostrar la respuesta.
- Ejemplo: En una aplicación de tienda online, cuando el usuario accede a la URL de un producto, el controlador gestiona esa solicitud, recupera los datos del producto desde el modelo y selecciona la vista que debe mostrar los detalles del producto.
- Tecnologías: Puede estar implementado en cualquier lenguaje del lado del servidor, como PHP, Python, Java, Ruby, o JavaScript con Node.js.

Flujo de Trabajo en MVC

1. El usuario interactúa con la vista, por ejemplo, haciendo clic en un botón o enviando un formulario.
2. La vista envía una solicitud al controlador. Este puede ser un enlace a una URL o un evento que desencadena una acción en el servidor.

3. El controlador procesa la solicitud. Interactúa con el modelo para obtener o modificar los datos según sea necesario.
4. El modelo realiza la lógica de negocio. Puede consultar una base de datos, realizar cálculos o aplicar reglas de negocio.
5. El controlador selecciona una vista y le pasa los datos necesarios para que el usuario vea el resultado de su acción.
6. La vista se actualiza con los datos obtenidos del controlador y muestra la respuesta al usuario.

Ejemplo práctico:

Imagina una aplicación web que muestra una lista de usuarios:

1. El usuario accede a una URL que muestra la lista de usuarios.
2. El controlador recibe la solicitud de esa URL.
3. El controlador pide al modelo que recupere todos los usuarios de la base de datos.
4. El modelo accede a la base de datos, obtiene la lista de usuarios y se la pasa al controlador.
5. El controlador pasa la lista a la vista.
6. La vista genera un HTML con los datos de los usuarios y lo muestra al usuario en el navegador.

Beneficios del patrón MVC:

- Separación de responsabilidades: Cada componente tiene una función clara, lo que facilita la comprensión y mantenimiento del código.
- Escalabilidad: Puedes modificar o añadir nuevas funcionalidades sin afectar otros componentes. Por ejemplo, cambiar la base de datos no afecta la vista.

- Reutilización de código: El modelo y el controlador pueden ser reutilizados con diferentes vistas o interfaces (web, móvil, API).
- Facilita el trabajo en equipo: Los desarrolladores pueden trabajar en capas diferentes sin interferencias (uno puede trabajar en el backend mientras otro se encarga del frontend).

Este patrón es muy popular en frameworks web como Ruby on Rails, Django (Python), Laravel (PHP), Spring (Java), entre otros.