INTRODUCCIÓN AL DESARROLLO BACKEND

CON NODEJS

2023

ANDÉN Centro de Innovación y Emprendimientos Tecnológicos

SECRETARÍA DE EXTENSIÓN UNIVERSITARIA UTN - FRC SEU

UTN Facultad Regional Córdoba

Agencia CÓRDOBA JOVEN

CÓRDOBA entre todos

Mejores prácticas en programación backend

- Arquitectura de aplicaciones backend: Patrones y enfoques.
  - Arquitectura Hexagonal
  - Patrón BFF.
  - Arquitectura de Capas
  - Modelo-Vista-Controlador (MVC)
  - Microservicios.
  - Arquitectura orientada a eventos.
- Consumo de APIs
  - Tipos de APIs
  - Realizando solicitudes HTTP en Node.js
  - Manejo de Respuestas
  - Prácticas recomendadas en el consumo de APIs
- Mejores prácticas en programación backend.
  - Principios SOLID.
  - Desarrollo Seguro
- Escalabilidad / Resiliencia Backend
  - Escalabilidad. Tipos de Escalabilidad
  - Resiliencia
  - Importancia de Escalabilidad y Resiliencia en Backend.

Arquitectura de aplicaciones backend: Patrones y enfoques

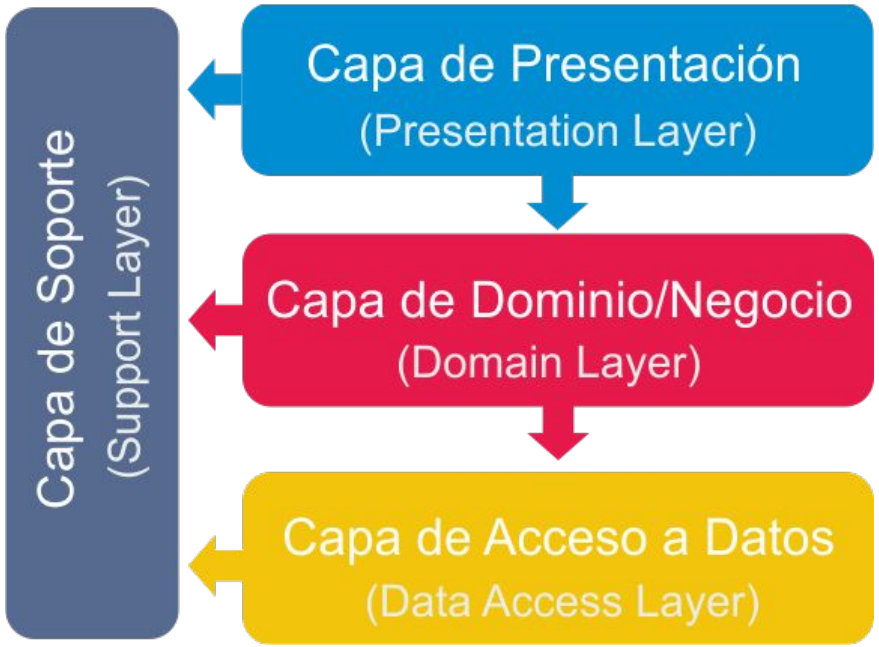
- La arquitectura de una aplicación backend es como el esqueleto de una casa.
- Define la estructura y organización del código para crear aplicaciones sólidas y escalables.
- Los patrones y enfoques nos guían para resolver problemas comunes y mantener un código de calidad.

Principales patrones arquitectónicos

En el desarrollo backend, hay varios patrones arquitectónicos comunes:

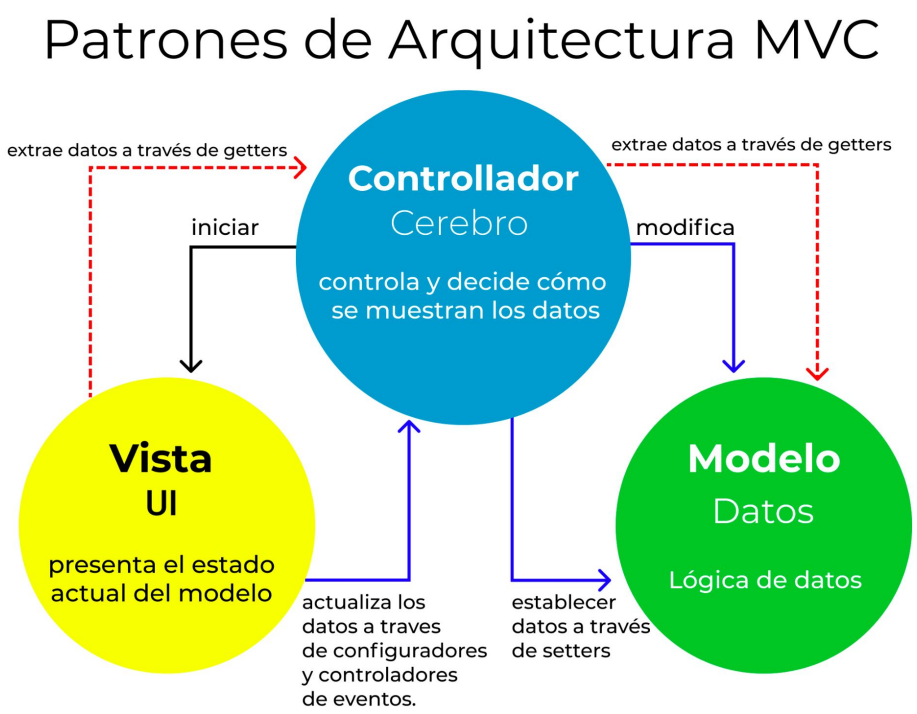
- Arquitectura de Capas
- Modelo-Vista-Controlador (MVC)
- Microservicios
- Arquitectura orientada a eventos
- BFF (Backend For Frontend)
- Arquitectura Hexagonal

Arquitectura de Capas



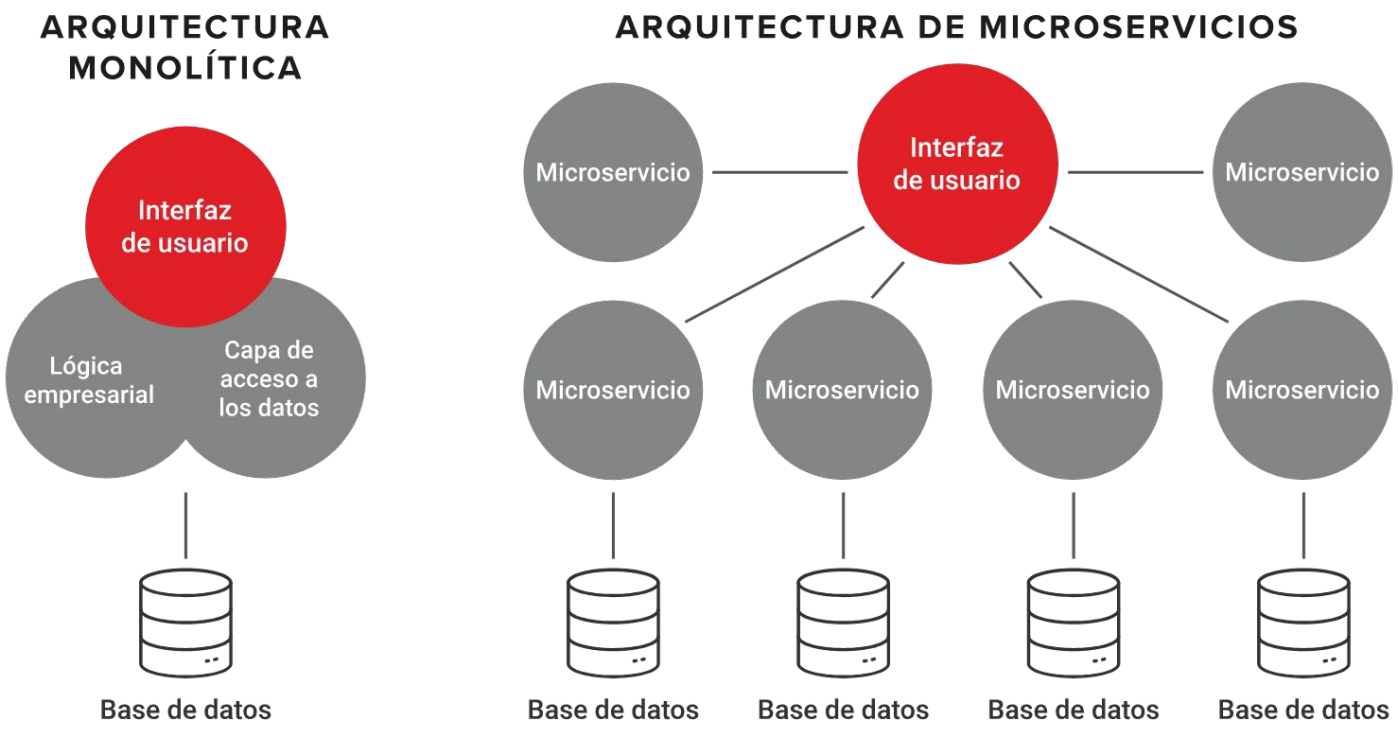
- El patrón de arquitectura de capas divide la aplicación en capas con responsabilidades específicas.
  - **Capa de Presentación:** Interfaz de usuario y lógica de presentación.
  - **Capa de Negocios:** Lógica de negocio y reglas del dominio.
  - **Capa de Datos:** Acceso y gestión de datos.

Modelo-Vista-Controlador (MVC)

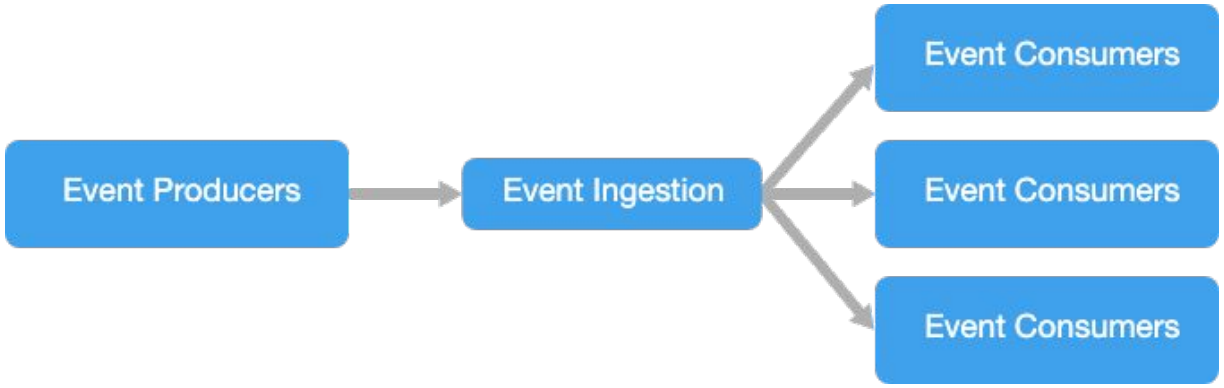


- El patrón Modelo-Vista-Controlador separa la aplicación en tres componentes principales.
  - **Modelo:** Representa los datos y la lógica de negocio.
  - **Vista:** Interfaz de usuario y presentación de datos.
  - **Controlador:** Gestiona las interacciones del usuario y coordina las acciones.

Arquitectura de Microservicios



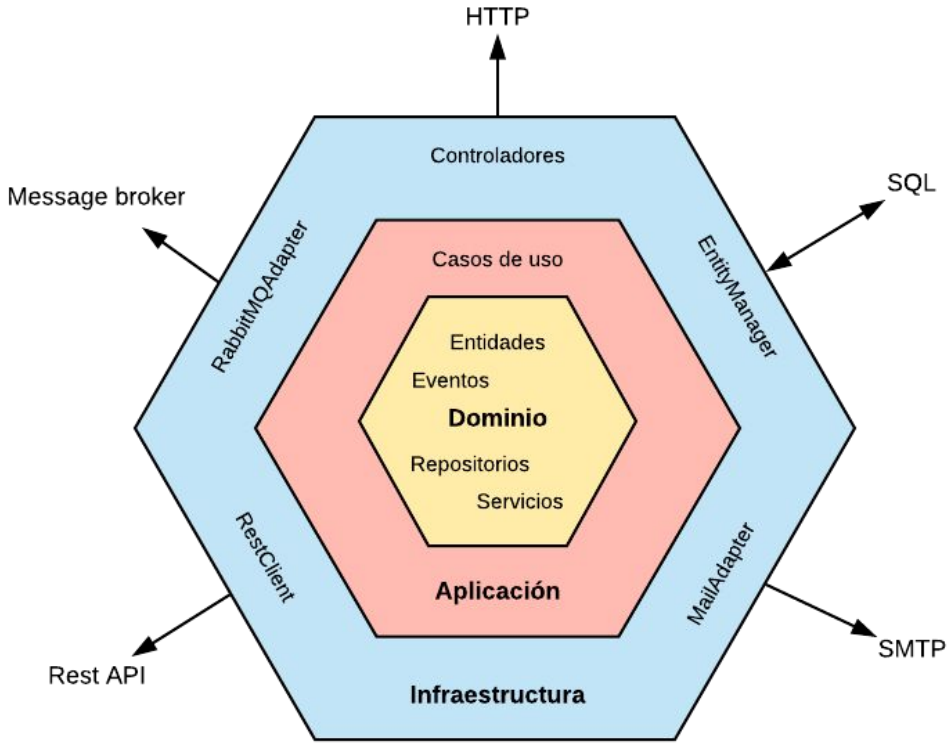
Arquitectura Orientada a Eventos



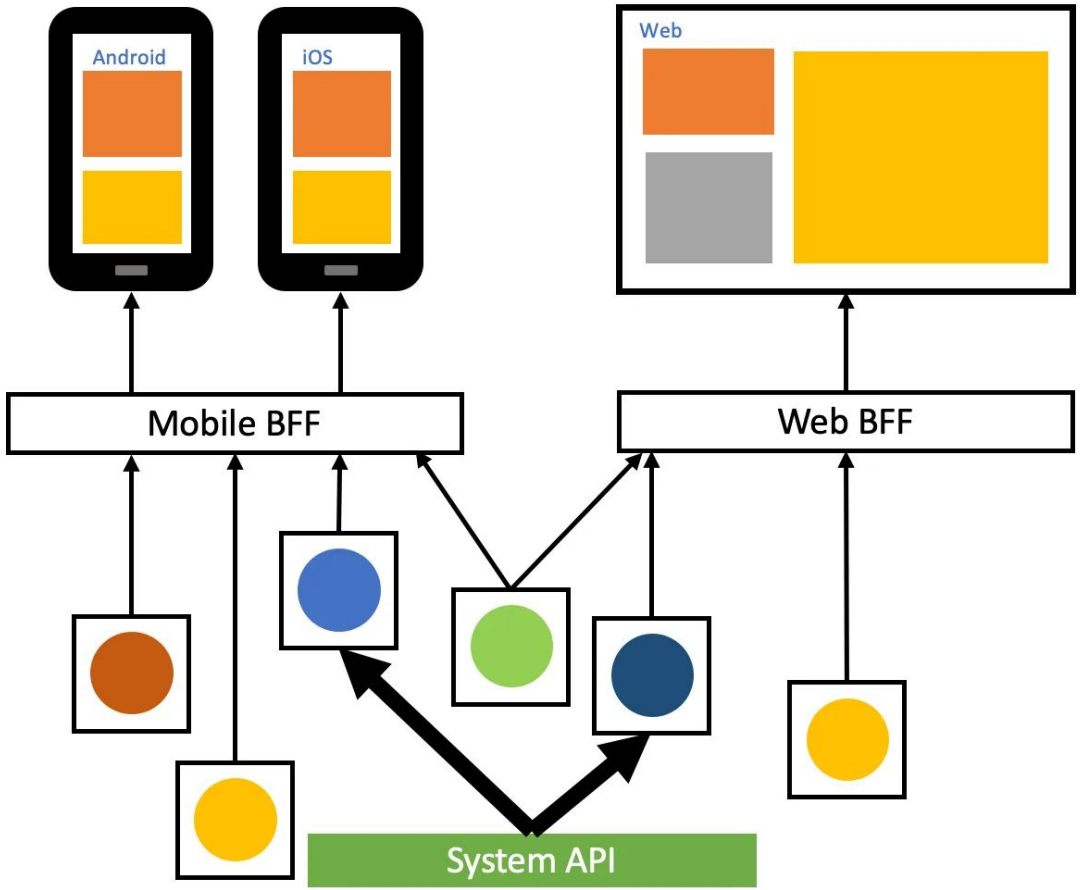
- La Arquitectura Orientada a Eventos se basa en el intercambio asíncrono de eventos entre componentes.
- Los eventos representan cambios de estado o acontecimientos significativos en la aplicación.
- Los componentes reaccionan y se actualizan en función de los eventos que reciben.

Arquitectura Hexagonal

- La Arquitectura Hexagonal, también conocida como Arquitectura de Puertos y Adaptadores, es un enfoque que busca separar la lógica de negocio del resto de la infraestructura.
- Se basa en tres componentes clave: Núcleo (lógica de negocio), Puertos (interfaces) y Adaptadores (implementaciones concretas).
- Proporciona una mayor flexibilidad, mantenibilidad y adaptabilidad de la aplicación.



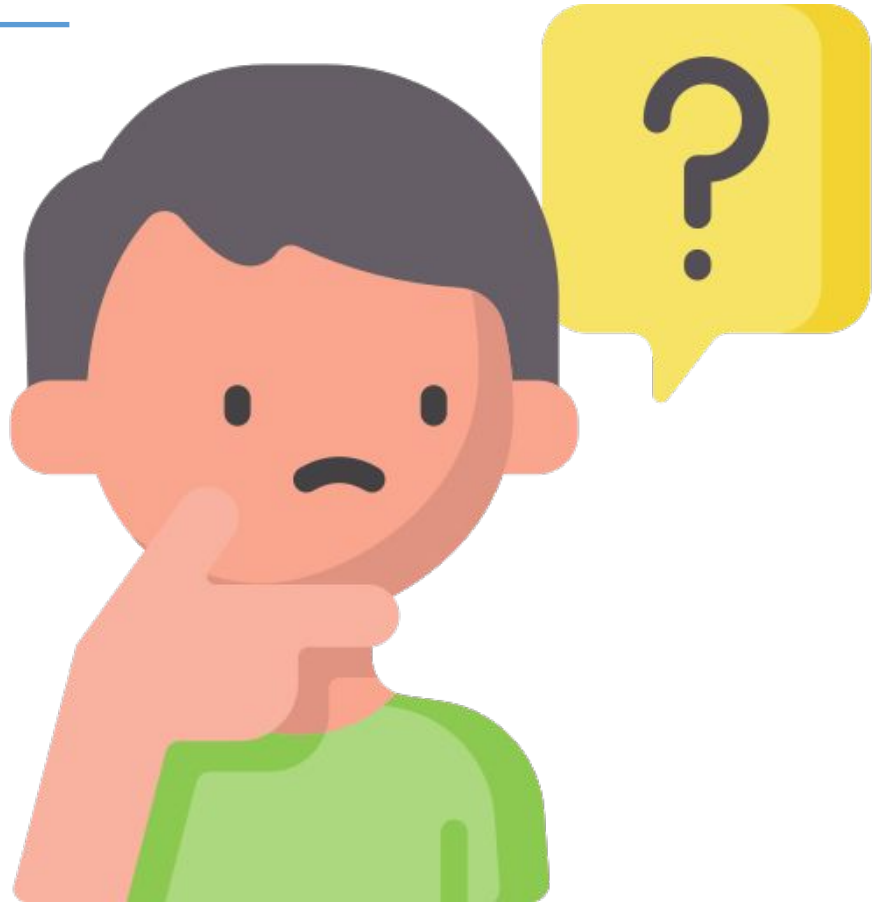
Backend For Frontend (BFF)



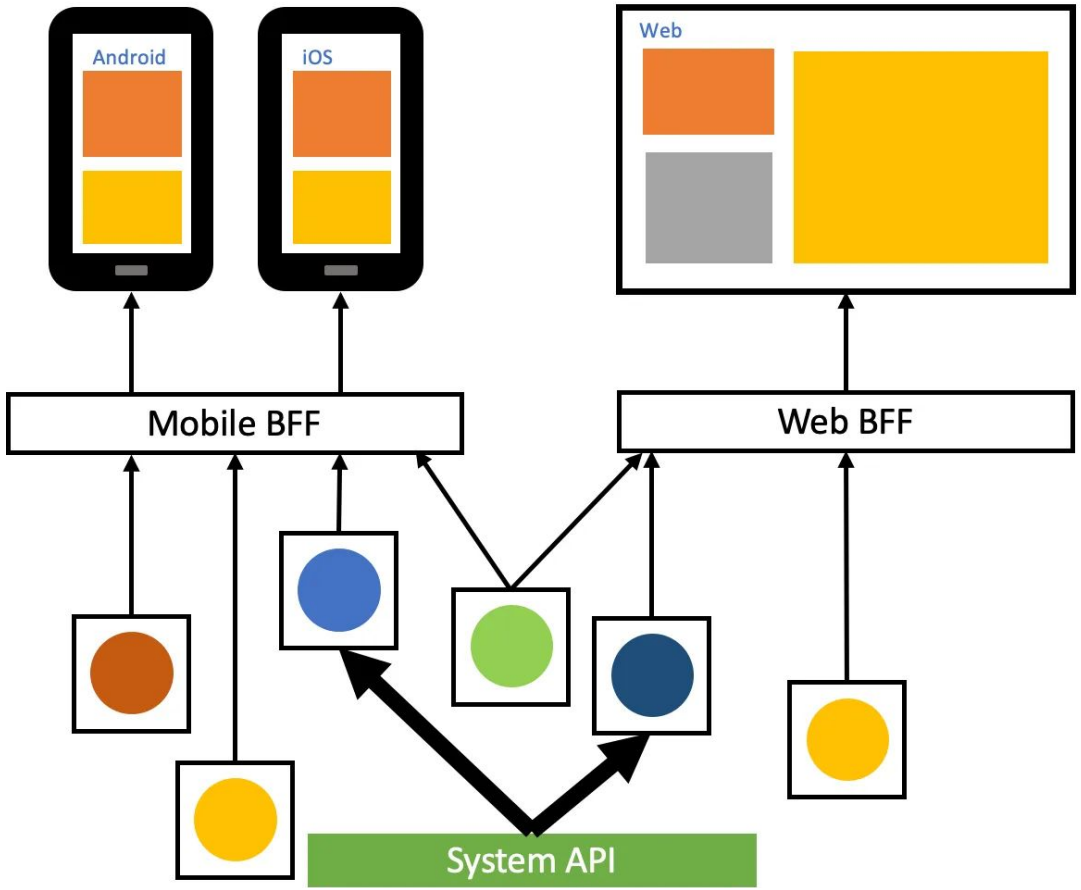
- Backend For Frontend (BFF) es un enfoque arquitectónico donde se crean backends especializados para satisfacer las necesidades específicas de cada interfaz de usuario o cliente.
- Cada interfaz tiene su propio BFF, lo que permite proporcionar datos y lógica de negocio específicos para esa interfaz.
- BFFs facilitan el desarrollo frontend, asegurando que las interfaces obtengan exactamente lo que necesitan sin exponer la lógica compleja del backend.

¿Cuál es el enfoque adecuado?

- La elección del enfoque arquitectónico depende de los requisitos del proyecto, el tamaño del equipo y las necesidades de escalabilidad y mantenibilidad.
- Es importante comprender cada enfoque y evaluar cómo se ajusta a los objetivos y restricciones del desarrollo de la aplicación.

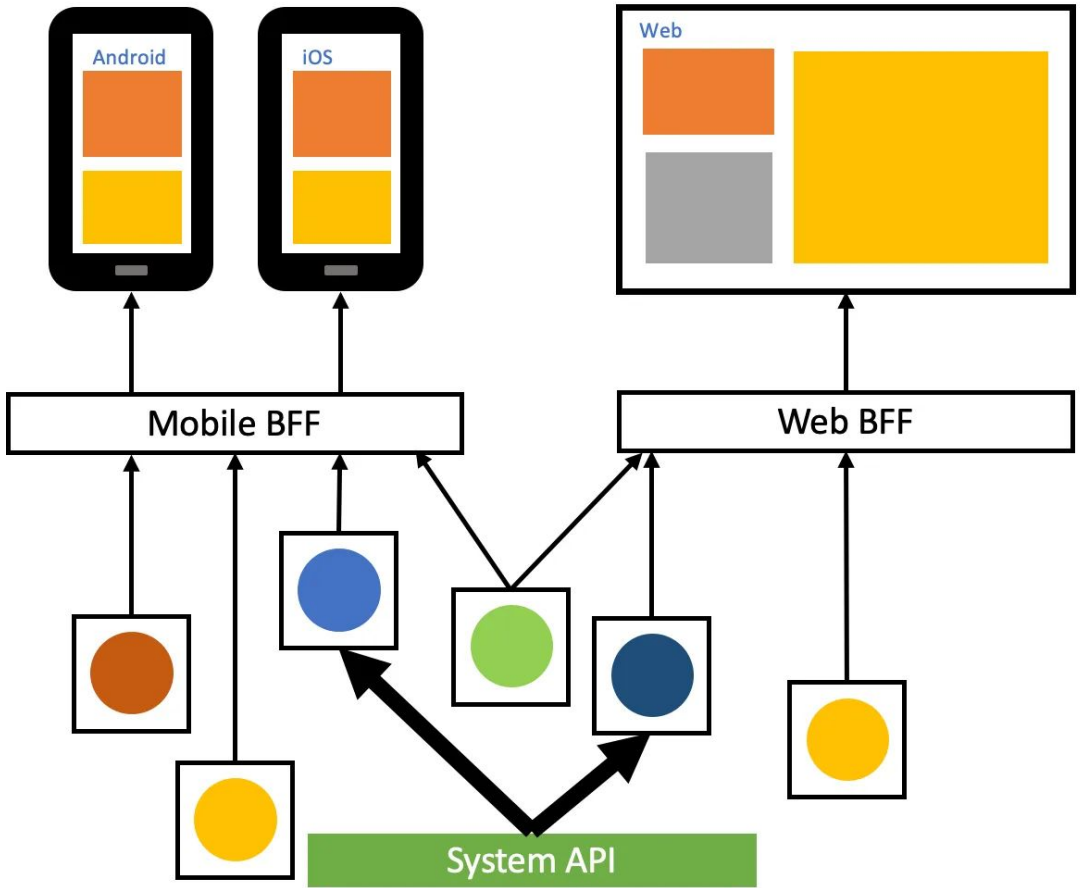


Consumo de APIs



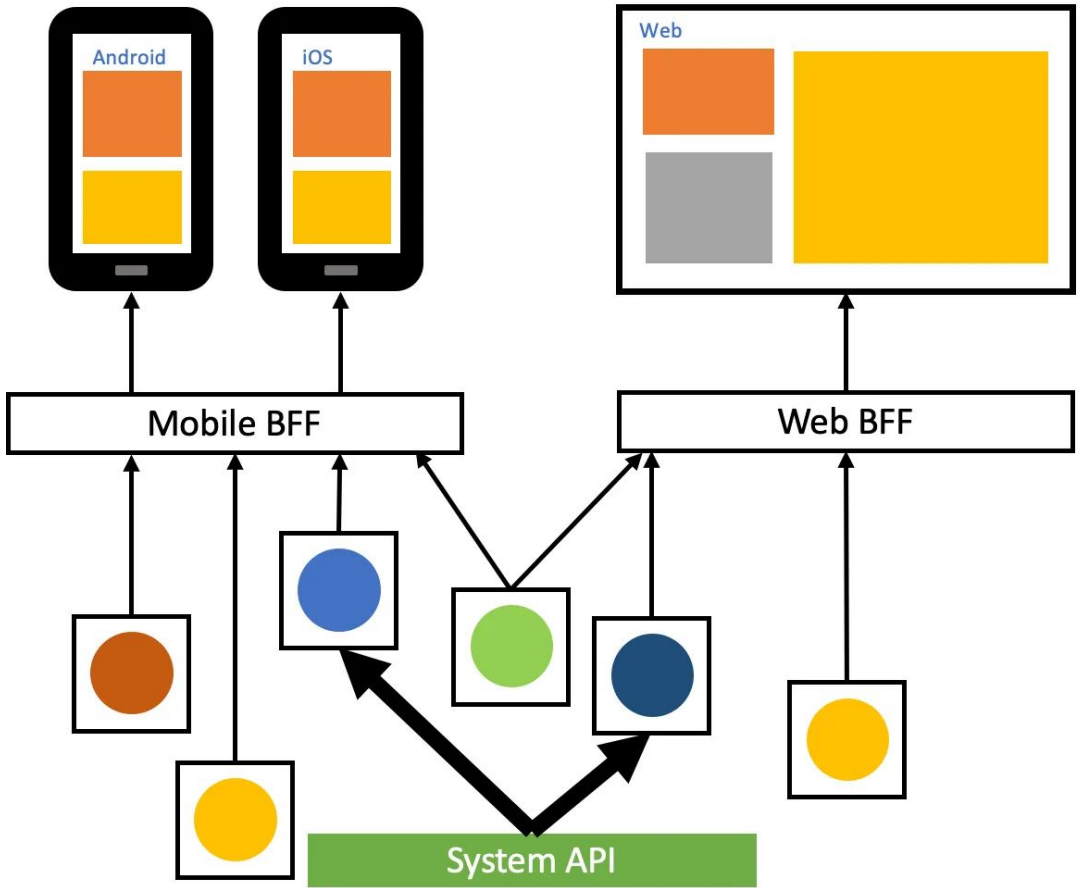
- **Solicitudes HTTP:**
  - Las APIs se consumen a través de solicitudes HTTP.
  - Cada solicitud tiene una URL única que identifica el recurso deseado.
  - Necesitamos un Cliente HTTP para consumir APIs.
- **Respuestas JSON:**
  - La mayoría de las APIs responden con datos en formato JSON (JavaScript Object Notation).
  - JSON es fácil de leer y manipular para las aplicaciones.
- **Autenticación y Claves de API:**
  - Algunas APIs requieren autenticación para proteger el acceso a sus recursos.
  - Se utilizan claves de API para autenticar la solicitud.
- **Manejo de Errores:**
  - Es importante manejar adecuadamente los errores que pueden ocurrir al consumir APIs.
  - Se pueden usar códigos de estado HTTP y mensajes de error para identificar problemas.

Tipos de APIs



- **APIs RESTful.**
- **GraphQL:** Lenguaje de consulta para APIs que permite a los clientes solicitar datos específicos en una sola consulta, proporcionando respuestas más eficientes y flexibilidad en el desarrollo.
- **gRPC:** Sistema de comunicación de código abierto basado en HTTP/2 y RPC, que utiliza Protocol Buffers para el intercambio de datos, permitiendo llamadas a métodos remotos eficientes y de alto rendimiento entre clientes y servidores.

Realizando solicitudes HTTP en Node.js



- Desde el Navegador Barra direcciones.
- Desde el Navegador con Javascript. AJAX.
- Desde otra API. Cliente HTTP.
- Desde Postman.

Librerías solicitudes HTTP en Node.js

- **axios:** Una librería HTTP basada en promesas que proporciona una interfaz fácil de usar y es ampliamente utilizada en el desarrollo de aplicaciones Node.js y JavaScript en general.
- **node-fetch:** Una librería moderna y liviana que implementa la API Fetch, que es estándar en los navegadores web, facilitando el consumo de APIs en Node.js.
- **request:** Una librería ampliamente conocida y utilizada que proporciona una API sencilla para realizar solicitudes HTTP.

Bueno, Vamo a Codea!!!

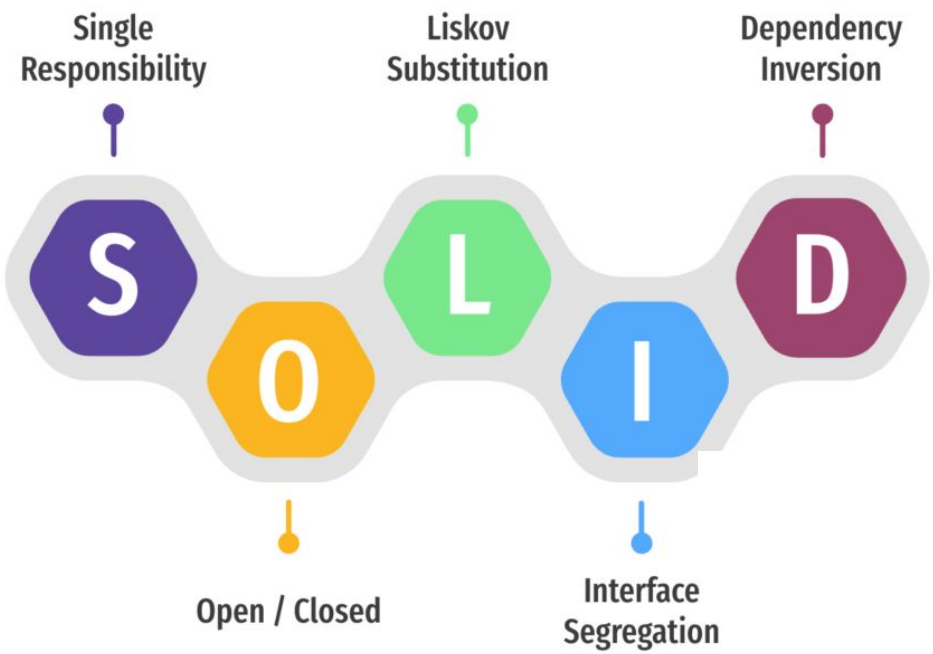
Con Axios vamos a crear un cliente HTTP desde una aplicación Nodejs.

AXIOS

Mejores prácticas en programación backend

- En el desarrollo backend, seguir mejores prácticas es esencial para construir aplicaciones sólidas, mantenibles y seguras.
- A continuación, exploraremos dos aspectos fundamentales:
  - Los principios SOLID
  - El desarrollo seguro.

Principios SOLID



- **Principio de Responsabilidad Única (Single Responsibility Principle):**
  - Cada clase debe tener una única responsabilidad o funcionalidad.
  - La separación de responsabilidades ayuda a facilitar el mantenimiento y la comprensión del código.
- **Principio de Abierto/Cerrado (Open/Closed Principle):**
  - El código debe estar abierto para la extensión pero cerrado para la modificación.
  - Se deben evitar modificaciones directas en el código existente para agregar nuevas funcionalidades.
- **Principio de Sustitución de Liskov (Liskov Substitution Principle):**
  - Las instancias de las subclases deben poder reemplazar a las instancias de las clases base sin alterar el comportamiento esperado.
- **Principio de Segregación de Interfaces (Interface Segregation Principle):**
  - Los clientes no deben verse obligados a depender de interfaces que no utilizan.
  - Las interfaces deben ser específicas y contener solo los métodos relevantes para su implementación.
- **Principio de Inversión de Dependencia (Dependency Inversion Principle):**
  - Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones.
  - La inversión de dependencia permite una mayor flexibilidad y facilita las pruebas unitarias.

Desarrollo Seguro

- **Validación de datos de entrada:**
  - Evitar ataques de inyección, como SQL injection y XSS (Cross-Site Scripting), validando y escapando los datos recibidos.
- **Autenticación y autorización adecuadas:**
  - Implementar un sólido sistema de autenticación para asegurar que solo usuarios autorizados accedan a los recursos.
- **Protección contra ataques comunes:**
  - Prevenir Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF) y otros ataques mediante medidas de seguridad adecuadas.
- **Registro y manejo de errores:**
  - Registrar errores para facilitar su identificación y resolución.
  - Implementar middleware de manejo de errores para centralizar y mejorar la gestión de errores.
- **Uso de HTTPS:**
  - Utilizar HTTPS para cifrar la comunicación entre el cliente y el servidor y proteger los datos sensibles.

Escalabilidad / Resiliencia Backend

- Escalabilidad y resiliencia son dos **características fundamentales** en el desarrollo de aplicaciones backend robustas y confiables.

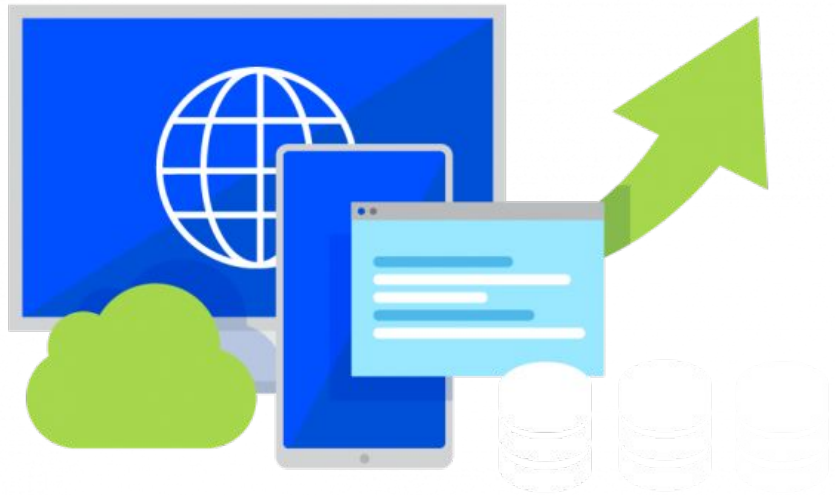
Escalabilidad

La escalabilidad se refiere a la capacidad de una aplicación para manejar un aumento en la carga de trabajo.

Permite que la aplicación crezca y se adapte a una mayor demanda de usuarios o procesamiento de datos.

Tipos de Escalabilidad

- Escalabilidad Vertical: Aumentar los recursos en una única máquina (por ejemplo, agregar más RAM o CPU).
- Escalabilidad Horizontal: Añadir más máquinas para distribuir la carga (escalabilidad mediante clustering o balanceo de carga).



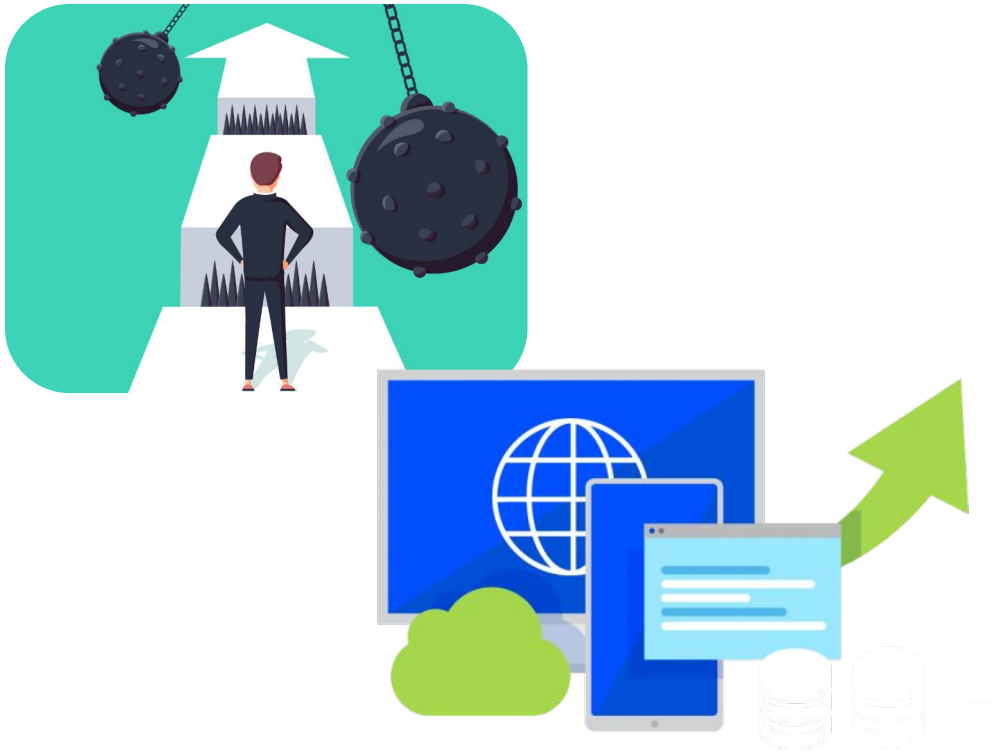
Resiliencia

La resiliencia se refiere a la capacidad de una aplicación para mantenerse en funcionamiento incluso ante fallos o situaciones adversas.

Técnicas para lograr Resiliencia

- Implementar redundancia y tolerancia a fallos en los sistemas y servicios críticos.
- Uso de patrones de diseño como Circuit Breaker y Retry para manejar errores de manera controlada.
- Realizar pruebas de resistencia y recuperación ante fallos.

Importancia de Escalabilidad y Resiliencia en Backend



- La escalabilidad y la resiliencia son esenciales para **enfrentar el crecimiento y la demanda** en aplicaciones en crecimiento.
- **Garantizan** que la aplicación se mantenga estable y confiable en todo momento.
- Escalabilidad y resiliencia son **pilares clave** en el desarrollo de aplicaciones backend exitosas.
- Proporcionan una **base sólida** para satisfacer las necesidades actuales y futuras de la aplicación.

MUCHAS TOTALES

ANDÉN Centro de Innovación y Emprendimientos Tecnológicos

SECRETARÍA DE EXTENSIÓN UNIVERSITARIA UTN - FRC SEU

UTN Facultad Regional Córdoba

Agencia CÓRDOBA JOVEN

CÓRDOBA entre todos