Arquitectura back-end para aplicaciones con IA

Breve descripción:

Este componente aborda las arquitecturas y prácticas esenciales para el desarrollo back-end de aplicaciones con inteligencia artificial. Explora los fundamentos de MVVM y persistencia de datos, la implementación de APIs RESTful y la integración de modelos de IA. Incluye diseño de interfaces con Jetpack Compose y metodologías de pruebas, proporcionando herramientas clave para construir aplicaciones robustas y escalables.

Diciembre 2024

Tabla de contenido

Introducción

La arquitectura back-end es el pilar fundamental en el desarrollo de aplicaciones que incorporan inteligencia artificial. Una arquitectura sólida garantiza el rendimiento y la escalabilidad del sistema, mientras facilita la integración eficiente de modelos de IA, permitiendo que las aplicaciones sean más inteligentes y responsivas.

Este componente aborda de manera sistemática los componentes clave de la arquitectura back-end para aplicaciones con IA, desde los fundamentos del patrón MVVM hasta las prácticas avanzadas en el desarrollo de APIs RESTful. Se exploran técnicas para la persistencia y gestión de datos, la implementación de modelos de IA en el servidor y la creación de interfaces de usuario modernas con Jetpack Compose.

A lo largo de estas páginas, se examinan metodologías actuales y prácticas comprobadas que permiten construir sistemas robustos y escalables. Se hace hincapié en la importancia de las pruebas y la documentación, reconociendo que una aplicación de calidad es el resultado de un código bien estructurado y exhaustivamente probado.

La combinación de conceptos teóricos con ejemplos prácticos proporciona las herramientas necesarias para enfrentar los desafíos reales en el desarrollo back-end de aplicaciones con inteligencia artificial. Como reza un principio fundamental en ingeniería de software: “Una aplicación es tan fuerte como su arquitectura subyacente”.

¡Le invitamos a emprender este viaje por las técnicas y prácticas esenciales de la arquitectura back-end para aplicaciones con IA!

Video 1. Arquitectura back-end para aplicaciones con IA

Enlace de reproducción del video

Síntesis del video: Arquitectura back-end para aplicaciones con IA
La arquitectura back-end es el cimiento sobre el cual se construyen aplicaciones innovadoras que incorporan inteligencia artificial. Este componente te guiará a través de las arquitecturas y prácticas esenciales para desarrollar sistemas robustos y escalables. Comenzaremos explorando los fundamentos del patrón MVVM y cómo su implementación, junto con herramientas como Retrofit y corrutinas, facilita la separación de responsabilidades y mejora la eficiencia en el consumo de APIs.

Luego, nos adentraremos en la persistencia y gestión de datos, desde el almacenamiento local con SharedPreferences y SQLite, hasta bases de datos externas y en tiempo real como MySQL y Firebase. Aprenderás a elegir y aplicar las estructuras de almacenamiento adecuadas según los requerimientos de tu aplicación.

El desarrollo de APIs RESTful es esencial para la comunicación entre el cliente y el servidor. Descubrirás cómo codificar APIs eficientes, integrar modelos de inteligencia artificial y utilizar herramientas clave como Retrofit y GSON para simplificar el proceso.

Finalmente, abordaremos el diseño de interfaces de usuario con Jetpack Compose, aplicando estilos y temas siguiendo las directrices de Material Design. También exploraremos las mejores prácticas en pruebas unitarias y documentación, asegurando la calidad y mantenibilidad de tus aplicaciones.

Este componente combina conceptos teóricos con ejemplos prácticos, proporcionándote las herramientas necesarias para enfrentar los desafíos reales en el desarrollo back-end de aplicaciones con inteligencia artificial.

¡Te invitamos a emprender este viaje al corazón de la arquitectura back-end para aplicaciones con IA!

1. Fundamentos de la arquitectura back-end para aplicaciones con IA

La arquitectura back-end es el cimiento sobre el cual se construyen aplicaciones sólidas y escalables que incorporan inteligencia artificial. Comprender sus fundamentos es esencial para desarrollar sistemas eficientes y mantenibles. En este capítulo, se explorarán los conceptos clave de la arquitectura Modelo-Vista-VistaModelo (MVVM), la importancia de los entornos de desarrollo integrados y los componentes básicos de las aplicaciones web.

1.1. Arquitectura MVVM

El patrón de arquitectura Modelo-Vista-VistaModelo, conocido como MVVM, es una metodología que promueve la separación de responsabilidades en el desarrollo de software. Este enfoque divide la aplicación en tres componentes principales: el Modelo, la Vista y el VistaModelo.

El Modelo representa los datos y la lógica de negocio de la aplicación. Es la capa donde se manejan las operaciones y reglas que definen el comportamiento del sistema. Por ejemplo, en una aplicación de predicción del clima, el Modelo gestionaría los datos meteorológicos y las reglas para interpretarlos.

La Vista es la interfaz de usuario. Es lo que el usuario ve y con lo que interactúa. La Vista presenta los datos del Modelo de forma visual y captura las interacciones del usuario, como clics o entradas de texto.

El VistaModelo actúa como un intermediario entre el Modelo y la Vista. Gestiona la lógica de presentación y maneja las comunicaciones entre la interfaz de usuario y la

lógica de negocio. El VistaModelo procesa las entradas del usuario, interactúa con el Modelo y actualiza la Vista en consecuencia.

Implementar MVVM ofrece varias ventajas. En primer lugar, permite una separación clara de responsabilidades, lo que facilita el mantenimiento y la escalabilidad de la aplicación. Al tener componentes independientes, es más sencillo realizar pruebas unitarias y detectar errores. Además, MVVM favorece la reutilización de código, ya que el VistaModelo puede ser utilizado por diferentes Vistas, y el Modelo puede servir a múltiples VistaModelos.

Para implementar MVVM en aplicaciones que interactúan con APIs, es común utilizar herramientas como Retrofit y corrutinas. Retrofit es una biblioteca que simplifica las llamadas HTTP, permitiendo consumir servicios web de forma eficiente. Por su parte, las corrutinas de Kotlin ofrecen una forma más sencilla y manejable de realizar operaciones asíncronas.

Al integrar Retrofit con corrutinas, se puede realizar llamadas a servicios web sin bloquear el hilo principal de la aplicación. Esto mejora la experiencia del usuario al mantener la interfaz de usuario receptiva mientras se realizan operaciones en segundo plano. Por ejemplo, una aplicación de mensajería puede utilizar corrutinas para enviar y recibir mensajes sin interrumpir la interacción del usuario con la interfaz.

La Clean Architecture complementa el patrón MVVM al proponer una estructura en capas que separa aún más las responsabilidades. Divide la aplicación en capas de Presentación, Dominio y Datos. La capa de Presentación incluye la Vista y el VistaModelo, la capa de Dominio contiene la lógica de negocio pura, y la capa de Datos maneja el acceso a fuentes de datos como bases de datos o servicios web.

La combinación de MVVM con Clean Architecture facilita la creación de aplicaciones modulares y mantenibles. Cada capa puede desarrollarse y probarse de forma independiente, lo que reduce la complejidad y mejora la calidad del software.

Tabla 1. Beneficios de MVVM y Clean Architecture

Beneficio	Descripción
Separación de responsabilidades.	Cada componente tiene una función clara, lo que facilita el mantenimiento y la escalabilidad.
Mejora en las pruebas.	La independencia de las capas permite realizar pruebas unitarias más efectivas.
Reutilización de código.	Los componentes pueden ser reutilizados en diferentes partes de la aplicación o en proyectos futuros.
Facilidad de mantenimiento.	Los cambios en una capa tienen un impacto mínimo en las demás, simplificando las actualizaciones.

Fuente. OIT, 2024.

1.2. Entornos de desarrollo integrado

El desarrollo de aplicaciones back-end para inteligencia artificial requiere de herramientas que faciliten la escritura, depuración y mantenimiento del código. Los entornos de desarrollo integrados (IDEs) proporcionan un conjunto de utilidades que mejoran la productividad y la calidad del software.

Entre los IDEs más utilizados se encuentran Visual Studio Code, Spyder-IDE y PyCharm. Visual Studio Code es un editor de código fuente ligero y extensible, compatible con múltiples lenguajes y plataformas. Ofrece características como

resaltado de sintaxis, autocompletado, depuración y una amplia gama de extensiones que pueden adaptarse a las necesidades del proyecto.

Spyder-IDE es un entorno de desarrollo específico para Python, orientado a científicos de datos y analistas. Incluye herramientas para la edición de código, ejecución interactiva y visualización de datos, lo que lo hace ideal para proyectos de inteligencia artificial y aprendizaje automático.

PyCharm, también enfocado en Python, es un IDE completo que ofrece potentes herramientas para el desarrollo web y científico. Proporciona soporte para frameworks populares, integración con sistemas de control de versiones y capacidades avanzadas de depuración y pruebas.

La estructura de un proyecto en estos entornos suele organizarse en módulos y archivos que reflejan la arquitectura de la aplicación. Los módulos agrupan funcionalidades relacionadas y facilitan la navegación y el mantenimiento del código. Por ejemplo, un módulo puede contener todas las clases y funciones relacionadas con la gestión de usuarios, mientras que otro maneja la interacción con la base de datos.

Los IDEs ofrecen vistas del proyecto que permiten explorar y gestionar los archivos y recursos de manera eficiente. Además, cuentan con ventanas de herramientas y paneles de navegación que proporcionan acceso rápido a funciones como la depuración, ejecución de pruebas, control de versiones y terminal integrada.

Conocer y aprovechar las funcionalidades del IDE seleccionado es fundamental para el desarrollo efectivo. Por ejemplo, utilizar atajos de teclado, configurar el entorno para el análisis estático del código y personalizar el editor puede ahorrar tiempo y reducir errores.

1.3. Conceptos básicos de aplicaciones web

Las aplicaciones web modernas están compuestas por varios componentes que interactúan para ofrecer funcionalidades complejas y una experiencia de usuario fluida. Entender estos componentes es esencial para desarrollar aplicaciones back-end que integren inteligencia artificial de manera efectiva.

Una aplicación suele constar de:

- **Actividades**: en el contexto de aplicaciones móviles y web, una actividad representa una pantalla o interfaz con la que el usuario interactúa. Gestiona el ciclo de vida de la interfaz y maneja eventos como la creación, pausa y destrucción.
- **Servicios**: son procesos en segundo plano que realizan tareas sin necesidad de interacción directa con el usuario. Por ejemplo, un servicio puede sincronizar datos con un servidor remoto o procesar información recibida.
- **Receptores de mensajes de distribución**: también conocidos como Broadcast Receivers, permiten que la aplicación responda a eventos del sistema o de otras aplicaciones. Pueden detectar cambios en la conectividad de red, recibir notificaciones de mensajes entrantes o reaccionar a alarmas programadas.
- **Proveedores de contenidos**: facilitan el intercambio de datos entre aplicaciones de manera segura y controlada. Permiten acceder y modificar datos almacenados en una aplicación desde otra, siempre que se tengan los permisos adecuados.

Estos componentes trabajan en conjunto para ofrecer funcionalidades complejas. Por ejemplo, una aplicación de noticias puede utilizar un servicio en segundo plano

para descargar los últimos artículos, un receptor de mensajes para notificar al usuario cuando hay contenido nuevo, y actividades para mostrar los artículos y permitir la interacción.

En el desarrollo back-end, es importante diseñar la interacción entre estos componentes de forma eficiente. La gestión adecuada de los servicios y receptores puede mejorar el rendimiento y la usabilidad de la aplicación. Además, garantizar la seguridad y privacidad en el intercambio de datos mediante proveedores de contenidos es esencial para proteger la información del usuario.

Cuando se integran componentes de inteligencia artificial, como modelos de aprendizaje automático, es común que el back-end deba manejar procesamiento intensivo de datos. Es fundamental optimizar estas operaciones para no afectar la experiencia del usuario. Utilizar técnicas de procesamiento asíncrono y manejar adecuadamente los hilos de ejecución ayuda a mantener la aplicación receptiva.

En resumen, este capítulo ha presentado los fundamentos esenciales para desarrollar una arquitectura back-end sólida en aplicaciones con inteligencia artificial. La comprensión y aplicación del patrón MVVM, complementado con la Clean Architecture, permite crear sistemas modulares y mantenibles. El uso de entornos de desarrollo integrados adecuados potencia la productividad y calidad del código.

Además, conocer los componentes básicos de las aplicaciones web y cómo interactúan es clave para diseñar soluciones eficientes y escalables. A medida que se avanza en el desarrollo de aplicaciones con inteligencia artificial, estos conceptos servirán como base para abordar desafíos más complejos y crear experiencias de usuario de alta calidad.

En los siguientes capítulos, se profundizará en temas como la persistencia y gestión de datos, el desarrollo de APIs RESTful y la creación de interfaces de usuario avanzadas. Estos conocimientos complementarán lo aprendido hasta ahora y permitirán construir aplicaciones completas que aprovechen al máximo las capacidades de la inteligencia artificial.

2. Persistencia y gestión de datos

La persistencia y gestión de datos son pilares fundamentales en el desarrollo de aplicaciones que incorporan inteligencia artificial. Un manejo eficiente y seguro de los datos garantiza no solo el correcto funcionamiento de la aplicación, sino también una experiencia de usuario óptima. En este capítulo, se explorarán las diversas opciones de almacenamiento, desde soluciones locales como SharedPreferences y SQLite, hasta bases de datos externas y en tiempo real como MySQL y Firebase. Además, se abordará la implementación de estructuras de almacenamiento y las mejores prácticas en la gestión de datos.

2.1. Persistencia de datos locales

SharedPreferences

SharedPreferences es una herramienta sencilla y eficiente que permite almacenar pares clave-valor de datos primitivos en el dispositivo del usuario. Es ideal para guardar configuraciones, preferencias del usuario o cualquier información pequeña que deba persistir entre sesiones de la aplicación.

Por ejemplo, si una aplicación necesita recordar el tema de color seleccionado por el usuario, puede almacenarlo utilizando SharedPreferences. Al reiniciar la aplicación, puede recuperar este valor y aplicar el tema correspondiente, mejorando así la experiencia del usuario.

Implementación básica de SharedPreferences:

Para guardar un valor:

val sharedPreferences = context.getSharedPreferences("MyPrefs", Context.MODE_PRIVATE)

```
val editor = sharedPreferences.edit()
editor.putString("tema_color", "oscuro")
editor.apply()
```

Para recuperar un valor:

```
val sharedPreferences = context.getSharedPreferences("MyPrefs", Context.MODE_PRIVATE)
val temaColor = sharedPreferences.getString("tema_color", "claro")
```

Bases de Datos SQLite: Conexión y Operaciones CRUD

Cuando se requiere manejar conjuntos de datos más complejos o voluminosos, SQLite es la base de datos local por excelencia en dispositivos móviles. Es una base de datos relacional ligera que permite realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) de manera eficiente.

Creación y configuración de la base de datos:

Para utilizar SQLite, se debe crear una clase que extienda de SQLiteOpenHelper y definir el esquema de la base de datos:

```
class MyDatabaseHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {
    override fun onCreate(db: SQLiteDatabase) {
```

```
        val createTable = "CREATE TABLE usuarios (id INTEGER PRIMARY KEY, nombre
TEXT, email TEXT)"
        db.execSQL(createTable)
    }
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        db.execSQL("DROP TABLE IF EXISTS usuarios")
        onCreate(db)
    }
}
```

Operaciones CRUD:

Crear (Insertar):

```
val db = dbHelper.writableDatabase
val values = ContentValues().apply {
    put("nombre", "Carlos Sánchez")
    put("email", "carlos@example.com")
}
db.insert("usuarios", null, values)
```

Leer (Consultar):

```
val db = dbHelper.readableDatabase
val cursor = db.query(
    "usuarios",
    arrayOf("id", "nombre", "email"),
    null,
    null,
    null,
    null,
    null
)
with(cursor) {
    while (moveToNext()) {
        val nombre = getString(getColumnIndexOrThrow("nombre"))
        val email = getString(getColumnIndexOrThrow("email"))
        // Procesar datos...
    }
}
cursor.close()
```

Actualizar:

```
val db = dbHelper.writableDatabase

val values = ContentValues().apply {

    put("email", "nuevo_email@example.com")

}

val selection = "id = ?"

val selectionArgs = arrayOf("1")

db.update("usuarios", values, selection, selectionArgs)
```

Eliminar:

```
val db = dbHelper.writableDatabase

val selection = "id = ?"

val selectionArgs = arrayOf("1")

db.delete("usuarios", selection, selectionArgs)
```

Manejo de multitablas

En aplicaciones más complejas, es común utilizar múltiples tablas relacionadas para representar entidades y sus relaciones. Por ejemplo, una aplicación de gestión de bibliotecas podría tener tablas para libros, autores y categorías.

Relaciones entre tablas:

- Uno a uno: un usuario tiene una única dirección.
- Uno a muchos: un autor ha escrito múltiples libros.
- Muchos a muchos: un libro puede pertenecer a varias categorías y una categoría puede incluir varios libros.

Para manejar estas relaciones, se utilizan claves primarias y foráneas, y se realizan consultas que combinan datos de múltiples tablas mediante JOINs.

Ejemplo de consulta con JOIN:

SELECT libros.titulo, autores.nombre

FROM libros

INNER JOIN autores ON libros.autor_id = autores.id

Esta consulta obtiene los títulos de los libros junto con el nombre de sus respectivos autores.

2.2. Bases de datos externas y en tiempo real

Conexión a MySQL: instalación y ejemplos

Para aplicaciones que requieren compartir datos entre múltiples usuarios o dispositivos, es necesario utilizar una base de datos externa como MySQL. MySQL es un sistema de gestión de bases de datos relacional ampliamente utilizado en aplicaciones web.

Instalación de MySQL:

- Descargar MySQL desde el sitio oficial.

- Instalar siguiendo las instrucciones proporcionadas, configurando el usuario y la contraseña de administrador.
- Configurar el acceso remoto si la aplicación necesita conectarse desde dispositivos externos.

Conexión desde la aplicación:

En aplicaciones móviles, no es recomendable conectarse directamente a la base de datos MySQL por motivos de seguridad y rendimiento. En su lugar, se utiliza un API RESTful que actúa como intermediario entre la aplicación y la base de datos.

Ejemplo de API RESTful en PHP:

```
<?php
header('Content-Type: application/json');
$conn = new mysqli("localhost", "usuario", "contraseña", "basedatos");
$result = $conn->query("SELECT * FROM usuarios");
$usuarios = array();
while($row = $result->fetch_assoc()) {
   $usuarios[] = $row;
}
echo json_encode($usuarios);
?>
```

La aplicación móvil puede consumir este API utilizando Retrofit:

```
interface ApiService {
    @GET("obtener_usuarios.php")
    suspend fun obtenerUsuarios(): List<Usuario>
}
```

Introducción a ORM y Room: configuración y uso

Los ORMs (Object-Relational Mapping) simplifican la interacción con bases de datos relacionales al permitir trabajar con objetos en lugar de tablas y consultas SQL. Room es el ORM oficial de Android para SQLite.

Configuración de Room:

Agregar dependencias en el archivo build.gradle:

```
implementation "androidx.room:room-runtime:2.4.0"
kapt "androidx.room:room-compiler:2.4.0"
```

Definir entidades (tablas):

```
@Entity(tableName = "usuarios")
data class Usuario(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val nombre: String,
```

```
    val email: String
)
```

Crear DAO (Data Access Object):

```
@Dao
interface UsuarioDao {
    @Insert
    suspend fun insertarUsuario(usuario: Usuario)



    @Query("SELECT * FROM usuarios")
    suspend fun obtenerUsuarios(): List<Usuario>
}
```

Configurar la base de datos:

```
@Database(entities = [Usuario::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun usuarioDao(): UsuarioDao
}
```

Uso de Room en la aplicación:

```
val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java, "mi_base_de_datos"
).build()
```

```
val usuarioDao = db.usuarioDao()
val nuevoUsuario = Usuario(nombre = "Laura López", email =
"laura@example.com")
usuarioDao.insertarUsuario(nuevoUsuario)
```

Firebase: conceptos, utilidades y servicios

Firebase es una plataforma de Google que ofrece servicios en la nube para facilitar el desarrollo de aplicaciones, incluyendo Firebase Realtime Database, una base de datos NoSQL en tiempo real.

Características principales:

- **Sincronización en tiempo real**: los datos se sincronizan automáticamente en todos los dispositivos conectados.

- **Soporte offline**: la aplicación sigue funcionando sin conexión y se sincroniza cuando vuelve a estar en línea.
- **Escalabilidad y seguridad**: maneja grandes volúmenes de datos con reglas de seguridad personalizables.

Implementación en la aplicación:

Agregar Firebase al proyecto: utilizando el asistente en Android Studio o mediante la consola de Firebase.

Escribir datos:

```
val database = Firebase.database

val myRef = database.getReference("usuarios")

myRef.child("usuario1").setValue(Usuario(nombre = "Ana Torres", email = "ana@example.com"))
```

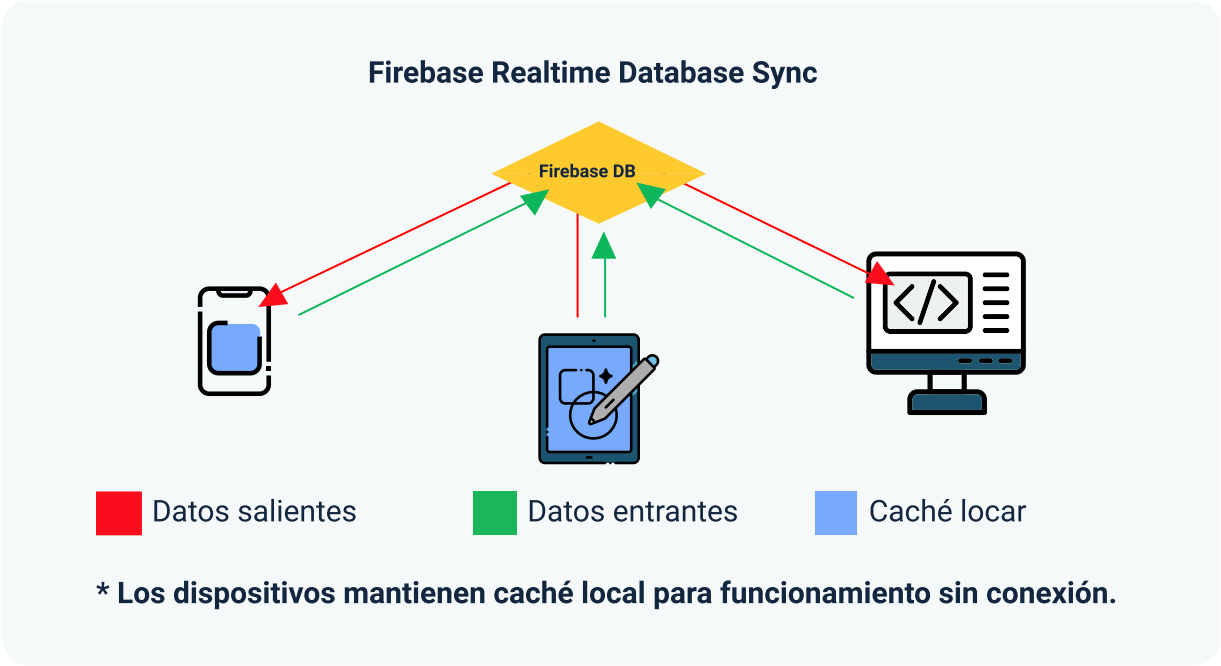
Leer datos:

```
myRef.addValueEventListener(object : ValueEventListener {
    override fun onDataChange(snapshot: DataSnapshot) {
        val usuarios = snapshot.children.mapNotNull { it.getValue(Usuario::class.java) }
        // Actualizar la interfaz de usuario
    }
```

```
    override fun onCancelled(error: DatabaseError) {
        // Manejar el error
    }
})
```

La siguiente figura ilustra cómo Firebase Realtime Database sincroniza datos entre múltiples clientes y el servidor en la nube. Cada cambio en los datos se refleja instantáneamente en todos los dispositivos conectados, lo que es ideal para aplicaciones colaborativas o que requieren actualización en tiempo real.

Figura 1. Arquitectura de Firebase Realtime Database



Fuente. OIT, 2024.

2.3. Implementación de estructuras de almacenamiento

Elegir la estructura de almacenamiento adecuada es indispensable para satisfacer los requerimientos específicos de cada aplicación. Factores como el tipo de datos, volumen, necesidad de sincronización y requisitos de seguridad influyen en esta decisión.

Empleo de estructuras de almacenamiento según requerimientos

- **SharedPreferences**: ideal para almacenar pequeñas cantidades de datos simples, como configuraciones o preferencias.
- **SQLite/Room**: adecuado para datos estructurados y relaciones complejas que deben almacenarse localmente.
- **MySQL**: útil para aplicaciones que requieren una base de datos centralizada y accesible desde múltiples dispositivos.
- **Firebase Realtime Database**: perfecto para aplicaciones que necesitan sincronización en tiempo real y funcionalidad offline.

Construcción de consultas para manipulación de información

Es esencial construir consultas eficientes y seguras para interactuar con las bases de datos. Al utilizar ORM como Room, muchas de estas preocupaciones se manejan automáticamente, pero siempre es importante seguir buenas prácticas.

Buenas prácticas en la construcción de consultas:

- **Utilizar consultas parametrizadas**: evita las inyecciones SQL y mejora la seguridad.
- **Optimizar las consultas**: seleccionar solo las columnas necesarias y utilizar índices apropiadamente.

- **Gestionar las transacciones**: asegurar la integridad de los datos en operaciones complejas.

Buenas prácticas en la gestión de datos

- **Validación de datos**: siempre validar los datos de entrada antes de almacenarlos o procesarlos.
- **Manejo de errores**: implementar mecanismos para manejar excepciones y errores en las operaciones de base de datos.
- **Seguridad y privacidad**: proteger los datos sensibles mediante encriptación y acceso controlado.
- **Backups y recuperación**: establecer estrategias de respaldo para prevenir la pérdida de datos.

En conclusión, la persistencia y gestión de datos son componentes esenciales en el desarrollo de aplicaciones con inteligencia artificial. Comprender las distintas opciones de almacenamiento y saber implementarlas adecuadamente permite crear aplicaciones robustas, eficientes y escalables. Al aplicar las mejores prácticas en la gestión de datos, se garantiza no solo el correcto funcionamiento de la aplicación, sino también la seguridad y satisfacción de los usuarios.

3. Desarrollo de APIs RESTful y modelos de IA

El desarrollo de APIs RESTful es fundamental para permitir la comunicación eficiente entre el cliente y el servidor en aplicaciones modernas, especialmente aquellas que integran inteligencia artificial. Este capítulo explora cómo codificar APIs RESTful, integrar modelos de IA en ellas y utilizar herramientas clave que facilitan el proceso de desarrollo.

3.1. Codificación de APIs RESTful

Las APIs RESTful (Representational State Transfer) son interfaces que permiten a las aplicaciones comunicarse con servicios web utilizando protocolos HTTP estándar. Al diseñar una API para un modelo de inteligencia artificial, es esencial seguir principios que garanticen su eficiencia, escalabilidad y facilidad de uso.

La creación de una API comienza con la definición clara de los recursos que se expondrán y las operaciones que se permitirán sobre ellos. Por ejemplo, si se desarrolla una aplicación que ofrece recomendaciones de películas basadas en preferencias del usuario, la API podría incluir recursos como "usuarios", "películas" y "recomendaciones". Cada recurso se manipula a través de métodos HTTP como GET, POST, PUT y DELETE, que corresponden a operaciones de lectura, creación, actualización y eliminación respectivamente.

Es importante adoptar estándares de codificación y patrones de diseño que aseguren la consistencia y mantenibilidad de la API. Utilizar convenciones de nomenclatura coherentes para las rutas y parámetros facilita la comprensión y uso por parte de otros desarrolladores. Además, implementar el manejo adecuado de errores y

códigos de estado HTTP permite que los clientes de la API gestionen las respuestas de manera efectiva.

Las pruebas son una parte integral del desarrollo de APIs. Herramientas como Postman y Hoppscotch permiten enviar solicitudes HTTP a la API y verificar las respuestas. Con ellas, es posible simular diferentes escenarios, probar la autenticación y asegurar que los `endpoints` funcionen correctamente bajo diversas condiciones. Estas herramientas también facilitan la documentación de la API, proporcionando ejemplos claros de cómo interactuar con ella.

3.2. Integración de modelos de inteligencia artificial

La incorporación de modelos de inteligencia artificial en una API añade un nivel de complejidad y potencia adicional. Estos modelos pueden ofrecer funcionalidades avanzadas como reconocimiento de voz, análisis de imágenes o predicciones basadas en datos.

Para implementar un modelo de IA en una API, primero es necesario entrenar el modelo utilizando datos relevantes y asegurarse de que su rendimiento cumple con los requisitos de la aplicación. Una vez entrenado, el modelo se integra en el servidor donde reside la API. Esto implica garantizar que las dependencias y bibliotecas necesarias estén disponibles en el entorno de ejecución.

Al diseñar el `endpoint` que expondrá las capacidades del modelo, se deben considerar los formatos de entrada y salida de datos. Por ejemplo, si el modelo analiza imágenes, el `endpoint` debe aceptar archivos de imagen en un formato específico y devolver resultados en un formato estructurado como JSON. Es importante también

manejar casos de error, como entradas no válidas o excepciones durante la ejecución del modelo.

Existen varios requerimientos y consideraciones clave al integrar modelos de IA en APIs:

- **Escalabilidad**: los modelos de IA pueden ser intensivos en recursos. Es esencial diseñar la API para que pueda escalar horizontalmente, distribuyendo la carga entre múltiples servidores si es necesario.
- **Latencia**: los tiempos de respuesta deben ser aceptables para los usuarios. Optimizar el rendimiento del modelo y utilizar técnicas como la caché pueden ayudar a reducir la latencia.
- **Seguridad**: al manipular datos potencialmente sensibles, es vital implementar medidas de seguridad robustas, como la autenticación y autorización adecuadas, y proteger contra ataques como la inyección de código.
- **Mantenimiento y actualización**: los modelos de IA pueden requerir actualizaciones periódicas para mantener su precisión. Es importante planificar cómo se desplegarán estas actualizaciones sin interrumpir el servicio.

Un ejemplo práctico de integración podría ser un servicio de traducción en tiempo real. El modelo de IA se entrena para traducir texto de un idioma a otro. La API expone un `endpoint` al que se envía texto en el idioma original y devuelve la traducción. Para garantizar un rendimiento óptimo, el modelo podría optimizarse utilizando técnicas de compresión o implementarse en un entorno de ejecución especializado como `TensorFlow Serving`.

3.3. APIs y herramientas para el desarrollo

El uso de herramientas y bibliotecas adecuadas puede simplificar significativamente el desarrollo y mantenimiento de APIs. En el contexto de aplicaciones móviles y web, Retrofit y GSON de Google son dos componentes esenciales.

Retrofit es una biblioteca para Android que facilita las solicitudes HTTP y la interacción con APIs RESTful. Permite definir interfaces en el código que representan los endpoints de la API, y luego, mediante anotaciones, especificar los métodos HTTP y rutas correspondientes. Retrofit maneja la serialización y deserialización de datos, lo que simplifica el manejo de las respuestas y solicitudes.

Spyder-IDE es un entorno de desarrollo específico para Python, orientado a científicos de datos y analistas. Incluye herramientas para la edición de código, ejecución interactiva y visualización de datos, lo que lo hace ideal para proyectos de inteligencia artificial y aprendizaje automático.

Por su parte, GSON es una biblioteca que permite convertir objetos Java en su representación JSON y viceversa. Al trabajar con Retrofit, GSON se utiliza para transformar automáticamente las respuestas JSON de la API en objetos que pueden manipularse en la aplicación. Esto reduce el código necesario para procesar los datos y minimiza errores asociados con el parsing manual.

Otra herramienta importante es la Cloud Storage API, que proporciona acceso a servicios de almacenamiento en la nube. Esto es útil cuando la aplicación necesita manejar archivos grandes o una cantidad significativa de datos que no es práctico

almacenar localmente. Integrar esta API permite cargar, descargar y gestionar archivos de manera segura y eficiente.

El EventBus es una biblioteca que facilita la comunicación entre diferentes componentes de la aplicación, especialmente en arquitecturas donde los componentes están desacoplados. Al publicar y suscribirse a eventos, se pueden notificar cambios o acciones sin crear dependencias directas entre clases. Esto mejora la modularidad y mantenibilidad del código.

La introducción a Android Pay (actualmente conocido como Google Pay) es relevante para aplicaciones que requieren procesar pagos. Integrar esta funcionalidad implica utilizar APIs específicas que manejan transacciones de manera segura, cumpliendo con estándares de seguridad y regulaciones financieras. Proporcionar una forma sencilla y confiable de realizar pagos mejora la experiencia del usuario y puede ser determinante para el éxito de la aplicación.

Tabla 2. Comparación de herramientas para el desarrollo de APIs

Herramienta	Función Principal	Beneficios
Retrofit.	Cliente HTTP para Android.	Simplifica solicitudes y respuestas.
GSON.	Serialización y deserialización JSON.	Manejo automático de datos.
Cloud Storage API.	Almacenamiento en la nube.	Gestión eficiente de archivos.
EventBus.	Comunicación entre componentes.	Desacoplamiento y modularidad.
Google Pay.	Procesamiento de pagos en la aplicación.	Seguridad y facilidad de uso.

Fuente. OIT, 2024.

En resumen, el desarrollo de APIs RESTful y la integración de modelos de inteligencia artificial son componentes esenciales en la creación de aplicaciones modernas y sofisticadas. Al seguir prácticas recomendadas en la codificación de APIs, utilizar herramientas adecuadas y considerar cuidadosamente los aspectos de integración de IA, es posible construir sistemas robustos, escalables y eficientes.

La capacidad de las APIs para conectar diferentes partes de una aplicación y permitir la interacción con servicios externos amplía significativamente las posibilidades de lo que se puede lograr. La inteligencia artificial añade una capa adicional de funcionalidad avanzada, permitiendo ofrecer experiencias personalizadas y soluciones inteligentes a problemas complejos.

Al continuar explorando estas tecnologías y perfeccionando las habilidades en su implementación, los desarrolladores pueden crear aplicaciones que no solo satisfagan las necesidades actuales de los usuarios, sino que también estén preparadas para adaptarse a futuros avances y demandas en el campo de la tecnología.

4. Desarrollo de interfaces y pruebas

El desarrollo de interfaces de usuario efectivas y la realización de pruebas exhaustivas son componentes esenciales en la creación de aplicaciones con inteligencia artificial. Una interfaz bien diseñada facilita la interacción del usuario con la aplicación, mientras que las pruebas aseguran que el sistema funcione correctamente y cumpla con los requisitos establecidos. En este capítulo, se explorará el uso de Jetpack Compose para el diseño de interfaces, la aplicación de estilos y temas siguiendo las directrices de Material Design, el desarrollo de aplicaciones web y las mejores prácticas en pruebas unitarias y documentación.

4.1. Diseño de interfaces con Jetpack Compose

Jetpack Compose es un moderno kit de herramientas de UI para Android que simplifica y acelera el desarrollo de interfaces de usuario. Basado en un enfoque declarativo, permite construir interfaces de manera más intuitiva y con menos código que los enfoques tradicionales.

Conceptos y características de Jetpack Compose

Jetpack Compose introduce una nueva forma de diseñar interfaces mediante funciones composables. Estas funciones describen la interfaz de usuario y su comportamiento, reaccionando automáticamente a los cambios en los datos subyacentes. Esto resulta en un código más limpio y fácil de mantener. Entre las características destacadas de Jetpack Compose se encuentran:

- **Programación declarativa**: se centra en qué mostrar en lugar de cómo mostrarlo.

- **Reactividad**: las UI se actualizan automáticamente cuando los datos cambian.
- **Compatibilidad**: se integra con vistas existentes y bibliotecas de Android.

Layouts: Box, Column y Row

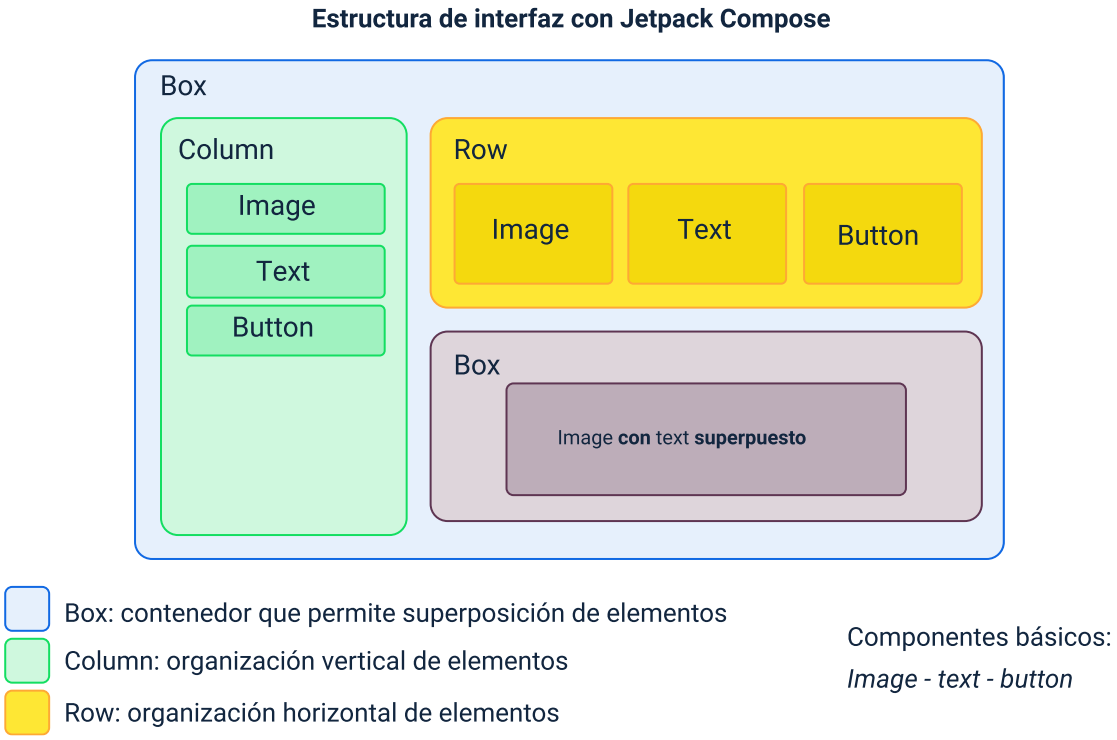
Los layouts son componentes fundamentales en el diseño de interfaces con Jetpack Compose. Permiten organizar y posicionar los elementos en la pantalla.

- **Box**: Es un contenedor que apila sus hijos uno encima del otro, permitiendo superposiciones.
- **Column**: Organiza los elementos en una disposición vertical, colocando cada elemento debajo del anterior.
- **Row**: Dispone los elementos horizontalmente, uno al lado del otro.

Estos layouts pueden anidarse y combinarse para crear estructuras más complejas. Por ejemplo, se puede utilizar una Column que contenga varias Rows para diseñar una cuadrícula personalizada.

La siguiente figura ilustra cómo se organizan los componentes en Jetpack Compose para crear una interfaz de usuario. Muestra la jerarquía de layouts y componentes, desde el contenedor principal hasta los elementos individuales.

Figura 2. Estructura de Jetpack Compose



Fuente. OIT, 2024.

Modificadores, textos e imágenes

Los modificadores son una herramienta poderosa que permite ajustar el comportamiento y apariencia de los componentes. Pueden cambiar el tamaño, el `padding`, el color, la alineación y más. El componente `Text` se utiliza para mostrar texto en la pantalla. Permite personalizar la tipografía, el estilo y aplicar modificadores para ajustar su presentación. Las imágenes se muestran utilizando el componente Image, que admite la carga de recursos locales o remotos. Se pueden aplicar modificadores para ajustar el tamaño, la forma y otros aspectos visuales.

Grids, Listas, Navegación y Tarjetas (Cards)

Para manejar listas de datos y estructuras repetitivas, Jetpack Compose ofrece componentes como LazyColumn y LazyRow, que renderizan eficientemente listas verticales y horizontales respectivamente.

Las grids permiten organizar elementos en forma de cuadrícula, ideal para galerías de imágenes o listas de productos. La navegación entre diferentes pantallas se gestiona mediante el componente NavHost, que define las rutas y destinos de la aplicación. Las tarjetas o Cards son componentes visuales que encapsulan contenido y acciones sobre un tema específico. Siguen las directrices de Material Design y aportan consistencia y estética a la aplicación.

4.2. Estilos, temas y material design

El diseño visual de una aplicación es determinante para ofrecer una experiencia de usuario atractiva y coherente. Aplicar estilos y temas permite unificar la apariencia y facilitar cambios globales en la interfaz.

a) Concepto de estilos y temas

Un estilo es un conjunto de atributos que definen la apariencia de un componente específico. Un tema es una colección de estilos que se aplican a una actividad o a toda la aplicación.

Utilizar estilos y temas ayuda a mantener la consistencia visual y simplifica la personalización de la interfaz. Por ejemplo, cambiar el color primario en el tema puede actualizar automáticamente todos los componentes que utilizan ese color.

b) Tipos de temas y prioridades

Existen diferentes niveles en los que se pueden aplicar temas y estilos:

- **Tema de aplicación**: afecta a todos los componentes de la aplicación.
- **Tema de activity**: se aplica a una actividad específica.
- **Tema de vista**: se aplica a un componente o vista en particular.
- **Estilo de vista**: define atributos para un tipo específico de componente, como botones o textos.
- **TextAppearance**: es un estilo específico para textos, que controla atributos como la fuente, el tamaño y el color.

La prioridad de aplicación de estilos y temas va desde los más generales (tema de aplicación) hasta los más específicos (estilo de vista). Esto permite sobrescribir atributos en niveles inferiores cuando sea necesario.

c) Componentes de Material Design: BottomAppBar, Floating Action Button, CardView

Material Design es un sistema de diseño desarrollado por Google que proporciona directrices para crear interfaces intuitivas y coherentes. Jetpack Compose incorpora muchos de estos componentes, facilitando su implementación.

- **BottomAppBar**: es una barra de navegación ubicada en la parte inferior de la pantalla, que puede contener acciones y elementos de navegación.
- **Floating Action Button (FAB)**: es un botón circular que representa la acción principal en una pantalla.

- **CardView**: es un contenedor que presenta contenido y acciones sobre un tema único, con sombras y bordes redondeados que le dan una apariencia elevada.

Utilizar estos componentes no solo mejora la estética de la aplicación, sino que también aporta familiaridad y facilita la navegación para el usuario.

4.3. Desarrollo y pruebas de aplicaciones web

Además del desarrollo de aplicaciones móviles, es importante considerar el desarrollo web para ofrecer soluciones multiplataforma.

a) Codificación en HTML5, CSS y JavaScript

El desarrollo de aplicaciones web se basa en tres tecnologías fundamentales:

- **HTML5**: define la estructura y el contenido de la página web.
- **CSS**: controla la presentación y el estilo de los elementos HTML.
- **JavaScript**: añade interactividad y lógica a la página.

Dominar estas tecnologías permite crear interfaces web dinámicas y responsivas. HTML5 introduce nuevas etiquetas semánticas que mejoran la accesibilidad y el SEO. CSS3 aporta características avanzadas de diseño, como animaciones y transiciones. JavaScript, junto con frameworks como React o Vue.js, facilita la creación de aplicaciones web complejas.

b) Elaboración y despliegue de interfaces web

El proceso de desarrollo incluye:

- **Diseño de la interfaz**: crear maquetas y prototipos para definir la apariencia y funcionalidad.
- **Codificación**: escribir el código HTML, CSS y JavaScript según el diseño.

- **Pruebas**: verificar el funcionamiento en diferentes navegadores y dispositivos.
- **Despliegue**: publicar la aplicación en un servidor web para que esté accesible a los usuarios.
- Pruebas de Integración con el Back-end

La interacción entre el front-end y el back-end es medular. Las pruebas de integración aseguran que las solicitudes y respuestas entre ambos funcionen correctamente.

Se utilizan herramientas como Postman para simular solicitudes desde el front-end y verificar que el back-end responda adecuadamente.

También se pueden implementar pruebas automatizadas que ejecuten escenarios completos de uso.

c) Corrección y optimización de parámetros

Después de las pruebas, es común identificar áreas de mejora. Esto puede incluir optimizar el rendimiento, corregir errores de lógica o mejorar la experiencia del usuario. Optimizar parámetros como el tamaño de las imágenes, minimizar el código CSS y JavaScript, y utilizar técnicas de carga diferida (lazy loading) puede mejorar significativamente la velocidad de la aplicación.

4.4. Pruebas Unitarias y Documentación

Las pruebas unitarias y la documentación son componentes esenciales para garantizar la calidad y mantenibilidad del software.

Configuración de entornos de prueba

Configurar un entorno de pruebas adecuado permite ejecutar pruebas en condiciones controladas. Esto incluye:

- **Entornos locales**: para pruebas iniciales y desarrollo.
- **Entornos de staging**: que replican el entorno de producción para pruebas más exhaustivas.

Generadores de Código de Pruebas

Existen herramientas que automatizan la creación de código de pruebas, facilitando la cobertura de casos y reduciendo el esfuerzo manual. Frameworks como JUnit para Java o PyTest para Python son ampliamente utilizados.

Las pruebas unitarias verifican el comportamiento de componentes individuales, asegurando que cada parte del código funcione como se espera.

Documentación y buenas prácticas

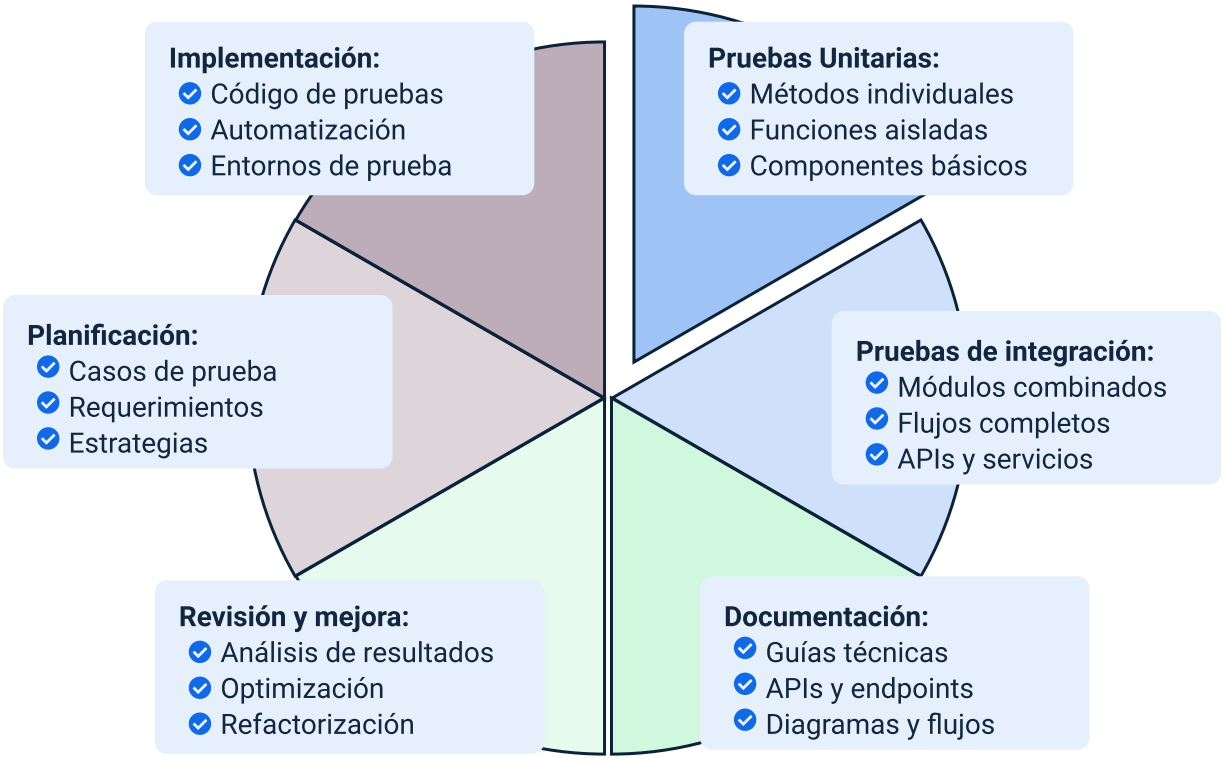
Mantener una documentación clara y actualizada es fundamental. Esto incluye:

- **Comentarios en el código**: para explicar la lógica y facilitar la comprensión.
- **Documentos técnicos**: que describen la arquitectura, las decisiones de diseño y cómo utilizar el software.
- **Manual de usuario**: orientado a los usuarios finales para ayudarles a entender y utilizar la aplicación.

Las buenas prácticas también implican seguir estándares de codificación, mantener un control de versiones adecuado y realizar revisiones de código.

La siguiente figura muestra el ciclo de pruebas y documentación en el desarrollo de software, destacando cómo las pruebas unitarias, de integración y la documentación interactúan para mejorar la calidad del producto.

Figura 3. Ciclo de pruebas y documentación



Fuente. OIT, 2024.

Nota: La infografía ilustra el flujo desde la escritura del código, pasando por las pruebas unitarias e integración, hasta la actualización de la documentación, enfatizando la naturaleza iterativa y continua de este proceso.

En esencia, el desarrollo de interfaces de usuario y la realización de pruebas son aspectos críticos en la creación de aplicaciones con inteligencia artificial. Utilizar

herramientas modernas como Jetpack Compose simplifica el diseño de interfaces atractivas y funcionales. Aplicar estilos y temas basados en Material Design asegura una experiencia de usuario coherente y agradable.

El desarrollo web complementa estas habilidades, permitiendo crear aplicaciones multiplataforma que alcancen una audiencia más amplia. Finalmente, las pruebas unitarias y la documentación son prácticas indispensables que garantizan la calidad y sostenibilidad del software a largo plazo.

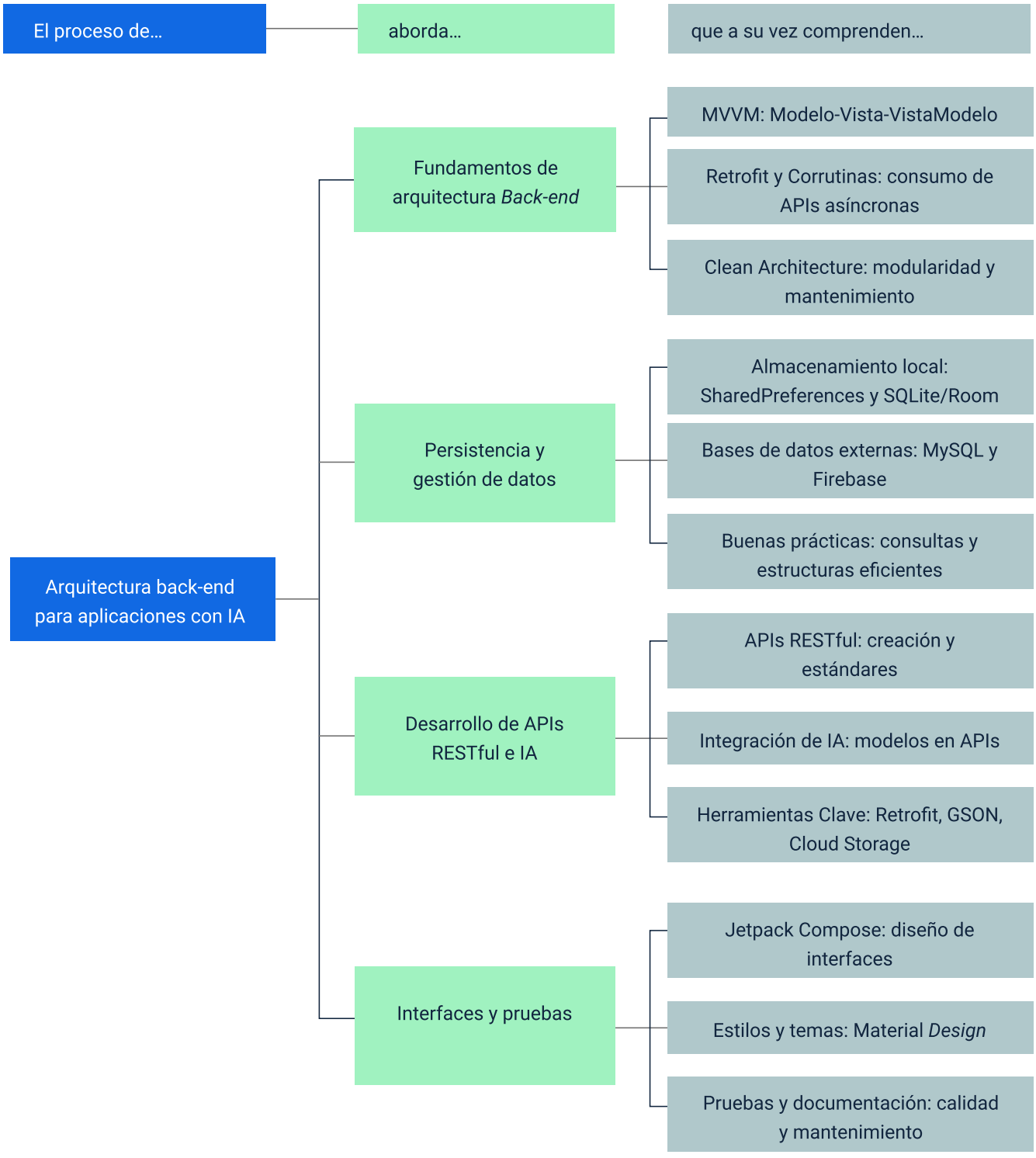
Al integrar estos conocimientos y técnicas, se puede desarrollar aplicaciones robustas, eficientes y centradas en el usuario, aprovechando al máximo las capacidades de la inteligencia artificial y ofreciendo soluciones innovadoras en un mercado cada vez más competitivo.

Síntesis

El diagrama representa la estructura integral del componente sobre arquitectura back-end para aplicaciones con inteligencia artificial. Partiendo del concepto central de arquitectura back-end, se ramifica en cuatro áreas esenciales: fundamentos de arquitectura, persistencia y gestión de datos, desarrollo de APIs RESTful e IA, e interfaces y pruebas. Cada área incorpora subtemas específicos que constituyen los elementos fundamentales para construir aplicaciones robustas y escalables que integran inteligencia artificial.

Esta organización ilustra el flujo lógico del proceso de desarrollo, desde la comprensión de patrones arquitectónicos como MVVM y Clean Architecture, pasando por la gestión eficiente de datos locales y en la nube, hasta la creación de APIs que incorporan modelos de IA. Finalmente, se aborda el diseño de interfaces modernas con Jetpack Compose y la importancia de las pruebas y documentación para asegurar la calidad y mantenibilidad del software.

El diagrama funciona como una hoja de ruta visual para comprender la estructura y alcance del componente, permitiendo al lector visualizar rápidamente la progresión del aprendizaje y las conexiones entre los diferentes temas. Se sugiere utilizarlo como referencia para organizar el estudio y entender cómo se integran los diversos aspectos en el desarrollo back-end de aplicaciones con inteligencia artificial.



Fuente. OIT, 2024.

Material complementario

Tema	Referencia	Tipo de material	Enlace del recurso
1. Fundamentos de la arquitectura back-end para aplicaciones con IA	Ecosistema de Recursos Educativos Digitales SENA. (2022b, octubre 5). Backend y Frontend.	Video	https://www.youtube.com/watch?v=ygzDmMLAoGk%3C
2. Persistencia y gestión de datos	Ecosistema de Recursos Educativos Digitales SENA. (2022a, julio 27). Arquitectura de tres niveles.	Video	https://www.youtube.com/watch?v=Y9a9__7QEfs
2. Persistencia y gestión de datos	Ecosistema de Recursos Educativos Digitales SENA. (2024, 24 septiembre). Desarrollo de aplicaciones web full stack.	Video	https://www.youtube.com/watch?v=wxE-QYhNtLc
2. Persistencia y gestión de datos	Ecosistema de Recursos Educativos Digitales SENA. (2023b, agosto 26). Gestión de bases de datos NoSQL con MongoDB.	Video	https://www.youtube.com/watch?v=60v7aYMVLyk
3. Desarrollo de APIs RESTful y modelos de IA	Ecosistema de Recursos Educativos Digitales SENA. (2023a, agosto 26). Construcción de API RESTful con Node.js.	Video	https://www.youtube.com/watch?v=GEgsJSgEPAk
4. Desarrollo de interfaces y pruebas	Ecosistema de Recursos Educativos Digitales SENA. (2022c, noviembre 18). Pruebas unitarias.	Video	https://www.youtube.com/watch?v=40b_UpuUbKo

Tema	Referencia	Tipo de material	Enlace del recurso
4. Desarrollo de interfaces y pruebas	Ecosistema de Recursos Educativos Digitales SENA. (2022d, noviembre 22). Ejemplo de prueba unitaria.	Video	https://www.youtube.com/watch?v=JtjLG40i-LI

Glosario

API RESTful: interfaz de programación que sigue los principios REST (Representational State Transfer). Utiliza métodos HTTP estándar como GET, POST, PUT y DELETE para permitir la comunicación y manipulación de recursos entre clientes y servidores.

Clean Architecture: enfoque de arquitectura de software que promueve la separación de responsabilidades en capas independientes. Mejora la mantenibilidad, escalabilidad y testabilidad al aislar la lógica de negocio de los detalles de implementación y frameworks.

Corrutinas: característica de Kotlin que permite escribir código asíncrono de manera secuencial. Facilita el manejo de operaciones que consumen tiempo, como solicitudes de red o acceso a bases de datos, sin bloquear el hilo principal de la aplicación.

Entorno de Desarrollo Integrado (IDE): aplicación que proporciona un conjunto completo de herramientas para el desarrollo de software en una sola interfaz. Incluye editor de código, depurador, compilador y otras utilidades que facilitan la programación.

EventBus: biblioteca para Android que implementa un sistema de publicación y suscripción de eventos. Permite la comunicación eficiente entre componentes de la aplicación (como actividades y fragmentos) sin crear dependencias directas, reduciendo el acoplamiento y mejorando la modularidad.

Firebase Realtime Database: base de datos NoSQL en tiempo real alojada en la nube de Firebase. Permite sincronizar y almacenar datos en formato JSON entre múltiples clientes de manera instantánea, con soporte para operaciones en línea y fuera de línea.

GSON: biblioteca de Java desarrollada por Google para convertir objetos Java en su representación JSON y viceversa. Facilita el manejo de datos al interactuar con servicios web que utilizan JSON como formato de intercambio.

Jetpack Compose: kit de herramientas moderno y declarativo para construir interfaces de usuario en Android. Simplifica el desarrollo de UI al permitir definir componentes de forma intuitiva y reactiva, actualizando automáticamente la interfaz ante cambios en los datos.

Material Design: sistema de diseño creado por Google que proporciona directrices, componentes y patrones visuales para crear interfaces de usuario consistentes, intuitivas y atractivas en aplicaciones móviles y web.

MVVM (Modelo-Vista-VistaModelo): un patrón de arquitectura de software que separa la aplicación en tres componentes: el Modelo (datos y lógica de negocio), la Vista (interfaz de usuario) y el VistaModelo (lógica de presentación que conecta el Modelo y la Vista). Facilita el mantenimiento y la escalabilidad de aplicaciones con interfaces complejas.

Pruebas Unitarias: tipo de pruebas de software que verifica el funcionamiento correcto de componentes individuales o unidades de código (como funciones o métodos). Aseguran que cada parte cumple con los requisitos y funciona aisladamente de otras partes del sistema.

Retrofit: biblioteca de cliente HTTP para Android y Java que simplifica las solicitudes y respuestas HTTP. Permite interactuar con APIs RESTful de manera eficiente, manejando automáticamente la serialización y deserialización de datos.

Room: biblioteca de persistencia de datos de Android que actúa como una capa de abstracción sobre SQLite. Simplifica las operaciones de base de datos utilizando objetos y anotaciones, facilitando el acceso a datos y reduciendo errores.

SharedPreferences: mecanismo de almacenamiento en Android que permite guardar datos simples en pares clave-valor. Es ideal para almacenar configuraciones, preferencias del usuario y pequeñas cantidades de información persistente.

SQLite: sistema de gestión de bases de datos relacional, liviano y de código abierto. Integrado en Android, permite almacenar y manipular datos estructurados localmente en el dispositivo sin necesidad de un servidor externo.

Referencias bibliográficas

Barbara, J. (2023). Practical C++ Backend Programming: Crafting Databases, APIs, and Web Servers for High-Performance Backend. GitforGits.

Borrego, A., Bermudo, M., Sola, F., Ayala, D., Hernández, I., & Ruiz, D. (2022). Silence — A web framework for an agile generation of RESTful APIs. SoftwareX, 20. https://doi.org/10.1016/j.softx.2022.101260

Firebase. (s. f.). Firebase Realtime Database. Firebase Documentation. Recuperado el 16 de octubre de 2023, de https://firebase.google.com/docs/database

Google Developers. (s. f.). Biblioteca de persistencia Room. Android Developers. Recuperado el 16 de octubre de 2023, de https://developer.android.com/training/data-storage/room

Google Developers. (s. f.). Guía de arquitectura de aplicaciones. Android Developers. Recuperado el 16 de octubre de 2023, de https://developer.android.com/jetpack/guide

Google Developers. (s. f.). Introducción a Jetpack Compose. Android Developers.

Google Developers. (s. f.). Pruebas de aplicaciones. Android Developers. Recuperado el 16 de octubre de 2023, de https://developer.android.com/training/testing

Herdiyatmoko, H. F. (2022). BACK-END SYSTEM DESIGN BASED ON REST API. Jurnal Teknik Informasi Dan Komputer (Tekinkom), 5(1), 123. https://doi.org/10.37600/tekinkom.v5i1.401

JetBrains. (s. f.). Guía de corrutinas de Kotlin. Documentación de Kotlin.

Kaluža, M., Kalanj, M., & Vukelić, B. (2019). A comparison of back-end frameworks for web application development. Zbornik Veleučilišta u Rijeci, 7(1), 317–332. https://doi.org/10.31784/zvr.7.1.10

Marquez-Soto, P. (2022). Backend Developer in 30 Days: Acquire Skills on API Designing, Data Management, Application Testing, Deployment, Security and Performance Optimization (English Edition). BPB Publications.

Martín, R. C. (2018). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.

Nurhayati, E., & Agussalim, A. (2023). Rancang Bangun Back-end API pada Aplikasi Mobile AyamHub Menggunakan Framework Node JS Express. Jurnal Sistem Dan Teknologi Informasi (JustIN), 11(3), 524. https://doi.org/10.26418/justin.v11i3.66823

Ormaechea, S., & Delgado, P. (2020). Desarrollo de aplicaciones con Android Jetpack. Marcombo.

Square, Inc. (s. f.). Retrofit. Square Open Source. Recuperado el 16 de octubre de 2023, de https://square.github.io/retrofit/

Vílchez, B., & Elizabeth, R. (2019). Arquitectura de back end con amazon web services (AWS) para sistemas escolares.

Créditos

Elaborado por:

Organización Internacional del Trabajo