Programacion III

Alejandro León Marín

October 12, 2024

Contents

1 Elementos esenciales del desarrollo de aplicaciones

El estándar ISO/IEC 9126 ha sido desarrollado en un intento de identificar los atributos clave de calidad para un producto de software. Este estándar es una simplificación del Modelo de McCall (Losavio et al., 2003), e identifica siete características básicas de calidad que pueden estar presentes en cualquier producto de software. Estas características son:

- **Funcionalidad:** Conjunto de atributos que relacionan la existencia de un conjunto de funciones con sus propiedades especificadas. Las funciones satisfacen necesidades especificadas o implícitas.

  Adecuación: Atributos que determinan si el conjunto de funciones son apropiadas para las tareas especificadas.

  Exactitud: Atributos que determinan que los efectos sean los correctos o los esperados.

  Interoperabilidad: Atributos que miden la habilidad de interactuar con sistemas especificados.

  Seguridad: Atributos que miden la habilidad para prevenir accesos no autorizados, ya sea accidentales o deliberados, tanto a programas como a datos.

- **Fiabilidad:**Conjunto de atributos que se relacionan con la capacidad del software de mantener su nivel de performance bajo las condiciones establecidas por un período de tiempo.

  Madurez: Atributos que se relacionan con la frecuencia de fallas por defectos en el software.

  Tolerancia a fallos: Atributos que miden la habilidad de mantener el nivel especificado de performance en caso de fallas del software.

  Recuperabilidad: Atributos que miden la capacidad de reestablecer el nivel de performance y recuperar datos en caso de falla, y el tiempo y esfuerzo necesario para ello.

  Cumplimiento: Atributos que hacen que el software se adhiera a estándares relacionados con la aplicación, y convenciones o regulaciones legales.

- **Usabilidad:** Conjunto de atributos que se relacionan con el esfuerzo necesario para usar, y en la evaluación individual de tal uso, por parte de un conjunto especificado o implícito de usuarios.

  Entendimiento: Atributos que miden el esfuerzo del usuario en reconocer el concepto lógico del software y su aplicabilidad.

  Aprendizaje: Atributos que miden el esfuerzo del usuario en aprender la aplicación (control, operación, entrada, salida).

Operabilidad: Atributos que miden el esfuerzo del usuario al operar y controlar el sistema.

- **Eficiencia:** Conjunto de atributos que se relacionan con el nivel de performance del software y la cantidad de recursos usados, bajo las condiciones establecidas.

  Comportamiento en tiempo: Atributos que miden la respuesta y tiempos de procesamiento de las funciones.

  Comportamiento en recursos: Atributos que miden la cantidad de recursos usados y la duración de tal uso en la ejecución de las funciones.

- **Mantenibilidad:** Conjunto de atributos que se relacionan con el esfuerzo en realizar modificaciones.

  Facilidad de análisis: Atributos que miden el esfuerzo necesario para el diagnóstico de deficiencias o causas de fallas, o para identificación de las partes que deben ser modificadas.

  Facilidad para el cambio: Atributos que miden el esfuerzo necesario para realizar modificaciones, remoción de fallas o cambios en el contexto.

  Estabilidad: Atributos que se relacionan con el riesgo de efectos no esperados en las modificaciones.

  Facilidad de pruebas: Atributos que miden el esfuerzo necesario para validar el software modificado.

- **Portabilidad:** Conjunto de atributos que se relacionan con la habilidad del software para ser transferido de un ambiente a otro.

  Adaptabilidad: Atributos que miden la oportunidad de adaptación a diferentes ambientes sin aplicar otras acciones que no sean las provistas para el propósito del software.

  Capacidad de instalación: Atributos que miden el esfuerzo necesario para instalar el software en el ambiente especificado.

  Conformidad: Atributos que miden si el software se adhiere a estándares o convenciones relacionados con portabilidad.

  Reemplazo: Atributos que se relacionan con la oportunidad y esfuerzo de usar el software en lugar de otro software en su ambiente.

- **Calidad de uso:** Conjunto de atributos relacionados con la aceptación por parte del usuario final y Seguridad.

  Eficacia: Capacidad de ayudar al usuario a realizar sus objetivos con exactitud y completitud.

  Productividad: Atributos relacionados con el rendimiento en las tareas cotidiana realizadas por el usuario final.

Satisfacción: Capacidad de satisfacer un usuario en un dado contexto de uso.

Seguridad: Capacidad de lograr aceptables niveles de riesgo para las personas, el ambiente de trabajo, y la actividad, en un dado contexto de uso.

2 Interfaces

2.1 Interfaces Funcionales

Las interfaces funcionales permiten pasar funciones como argumentos a métodos, algo que ya existía en Java (por ejemplo, con la interfaz Comparator), pero que fue ampliado en Java 8 con nuevas interfaces y aplicaciones. Estas interfaces se utilizan para definir comportamientos específicos. mediante métodos abstractos únicos, que pueden implementarse de manera más simple y concisa con expresiones lambda o referencias a métodos.

2.1.1 Expresiones Lambda

Una de las principales mejoras de Java 8 es la introducción de las lambda expresiones, que permiten una forma más compacta y legible de definir el comportamiento funcional. Las lambda expresiones se utilizan para simplificar la definición de funciones en lugar de crear clases o instancias anónimas.

2.1.2 Interfaces Funcionales Comunes

- **Predicate:** Evalúa una condición sobre un objeto de tipo T y devuelve un valor booleano. Ejemplo: determinar si un vuelo está completo.

```
Predicate<Vuelo> vueloCompleto =
x-> x.getNumPasajeros().equals(x.getNumPlazas());
```

- **BiPredicate ¡T,U¿:** Evalúa una condición sobre dos objetos de tipo T y U y devuelve un booleano. Ejemplo: verificar si un vuelo sale en una fecha específica.

```
BiPredicate<Vuelo, Fecha> getCoincidencia = (x,y)->
y.equals(x.getFecha());
```

- **Function ¡T,R¿:** Toma un argumento de tipo T y devuelve un resultado de tipo R. Se utiliza para transformar objetos. Ejemplo: obtener la duración de un vuelo.

```
Function<Vuelo, Duracion> functionDuracion = x->x.getDuracion();
```

- **BiFunction ¡T,U,R¿:** Toma dos argumentos de tipos T y U, y devuelve un resultado de tipo R

```
ToIntBiFunction<Vuelo,Fecha> getdias(Vuelo v, Fecha f)
return (x,y)->y.resta(x.getFecha());
```

- **Consumer ¡T¿:** Realiza una operación sobre un objeto de tipo T sin devolver un valor. Ejemplo: incrementar el precio de un vuelo.

```
Consumer<Vuelo> incrementaPrecio10p =
x->x.setPrecio(x.getPrecio()*1.1);
```

- **BiConsumer ¡T,U¿:** Realiza una operación sobre dos objetos de tipo T y U.

```
BiConsumer<Vuelo, Fecha> setFecha = (x,y)->x.setFecha(y);
```

- **Supplier ¡T¿:** Proporciona un objeto de tipo T sin recibir ningún argumento. Es útil para generar valores o instancias.

```
Supplier<Vuelo> dameVuelo = ()-> new VueloImpl();
```

- **UnaryOperator ¡T¿:** Representa una operación sobre un único operando de tipo T y devuelve un resultado del mismo tipo.

```
public UnaryOperator<Duracion> anadeMinutos(Integer m)
return x -> x.suma(new DuracionImpl(0,m));
```

- **BinaryOperator ¡T¿:** Representa una operación sobre dos operandos de tipo T y devuelve un resultado del mismo tipo.

```
public Duracion suma(Duracion d) {
        Integer min = getMinutos() + d.getMinutos();
        Integer hor = getHoras() + d.getHoras();
        return new DuracionImpl(hor+min/60,min%60);
}
```

En resumen, el documento explica cómo las interfaces funcionales y las lambda expresiones permiten escribir código más flexible y reutilizable en Java 8, facilitando la implementación de funciones que pueden ser pasadas como parámetros a métodos y aplicadas a objetos de manera más concisa y legible.

2.2 Interfaces Comparator y Comparable

2.2.1 Comparable

La interfaz Comparable permite definir un orden natural para los objetos de una clase mediante la implementación del método compareTo(Object o).

```
public interface Comparable<T> {
int compareTo(T o);
}
```

2.2.2 Comparator

La interfaz Comparator, en cambio, permite definir diferentes órdenes personalizados mediante la implementación del método compare(Object o1, Object o2). Es útil cuando se desea ordenar los objetos por criterios distintos al orden natural.

Se utiliza para comparar dos objetos de una colección. Es útil cuando queremos ordenar objetos de distintas formas. Por ejemplo, en una clase Persona, el orden natural podría basarse en el nombre (implementado con Comparable), pero si queremos ordenar por altura, debemos usar Comparator. El método compare(Object o1, Object o2) en Comparator permite especificar las reglas de comparación según el criterio deseado (por ejemplo, altura o fecha de nacimiento).

```
public interface Comparator<T> {
int compare(T o1, T o2);
}
```

Una conclusión puede ser que aunque parecen iguales Comparable y Comparator, en realidad no lo son y mientras una se define para el orden natural, la otra se define para un orden total respectivamente. El orden natural es utilizado por diversos métodos del api de Java, pero no siempre hemos de usar el orden natural. En determinados casos podremos querer ordenar objetos por un orden distinto al orden natural, y para ello nos será útil implementar la interface Comparator. La segunda conclusión es que gracias a la interfaz Comparator, podemos ordenar muy fácilmente colecciones utilizando clases que implementen el método compare por cada tipo de ordenación que deseemos.

2.3 Metodos default en Interfaces

Las interfaces en versiones anteriores solo podían contener métodos abstractos y constantes. Las clases que implementaban una interfaz estaban obligadas a proporcionar la implementación de todos los métodos declarados en ella. Si se añadía un nuevo método a la interfaz, las clases que la implementaban debían actualizarse para evitar errores de compilación.

A partir de Java 8, es posible declarar métodos con una implementación por defecto directamente en la interfaz, llamados métodos default. Estos métodos no requieren que las clases que implementan la interfaz los redefinan, ya que pueden ser utilizados directamente sin necesidad de implementación en la clase. A pesar de que los métodos por defecto tienen una implementación base, las clases que implementan la interfaz pueden sobreescribirlos para proporcionar un comportamiento específico si así lo desean.

En el ejemplo que se menciona, la interfaz ICalculadora incluye el método multiplicar como un método por defecto. La clase que implementa esta interfaz puede usar este método sin necesidad de implementarlo en su código.

Sin embargo, si la clase desea un comportamiento distinto para la multiplicación, puede sobreescribir este método por defecto.

3 Java versions

3.1 Java 8

Las caracteristicas de Java 8 son:

- **Métodos default en interfaces:** Ahora las interfaces pueden tener métodos con implementación por defecto, permitiendo que las clases que implementan la interfaz no necesiten definir esos métodos.
- **Interfaces funcionales y lambdas:** Las interfaces funcionales permiten definir funciones que pueden ser instanciadas por lambdas. Se introduce la anotación @FunctionalInterface para marcar una interfaz como funcional.
- **Expresiones Lambda:** Simplifican la declaración de funciones y reducen la verbosidad al definir comportamientos de métodos abstractos de las interfaces funcionales.
- **Predicate¡T¿:** Implementa una condición lógica que evalúa un objeto de tipo T devolviendo un booleano.
- **Stream API:** Introduce una nueva forma de trabajar con flujos de datos a través de colecciones, permitiendo realizar operaciones funcionales como filter(), map(), collect() y forEach().
- **java.time:** Un nuevo paquete que introduce clases para manejar fechas y tiempos de manera más eficiente, como LocalDate, LocalTime, Instant, y ZonedDateTime.

3.2 Java 9 - 11

3.2.1 Java 9

- **Modularidad (Proyecto Jigsaw):** Introduce la modularización del JDK, permitiendo una mejor encapsulación, menor superficie de ataque, dependencias explícitas y optimización de recursos. Mejora la organización de los paquetes en módulos que permiten definir dependencias entre ellos.
- **Mejoras en colecciones:** Nuevos métodos factoría para crear colecciones de manera más eficiente.
- **Mejoras en la clase Optional:** Métodos como or(), ifPresentOrElse(), y stream() mejoran su manejo.
- **Streams:** Nuevos métodos como dropWhile() y takeWhile(), además de ofNullable() y mejoras en iterate().
- **jlink:** Permite generar runtimes mínimos solo con los módulos necesarios, útil en contenedores y entornos cloud.

- **Recolector de basura G1:** Ahora es el recolector por defecto, optimizado para reducir la latencia.
- **Jar multiversión:** Soporte para archivos jar que contienen diferentes versiones de clases para distintas versiones de Java.

3.2.2 Java 10

Inferencia de tipos: Introducción de la palabra clave var para la inferencia de tipos en variables locales, simplificando la creación de objetos.

3.2.3 Java 11

Versión LTS: Proporciona soporte extendido hasta 2023 (o 2026 en forma extendida). Se destacan mejoras incrementales como el soporte para funciones avanzadas de seguridad y mejoras en el rendimiento.

3.3 Java 12 - 17

3.3.1 Java 12

- **Expresiones switch:** Introducción de manera experimental. Permite utilizar switch como expresión, mejorando la legibilidad del código.
- **Formato de número compacto:** Permite formatear números en un formato más compacto. Ejemplo: 1000 se convierte en 1K.

3.3.2 Java 13

- **Bloques de texto:** Introducción de bloques de texto que permiten escribir múltiples líneas sin necesidad de concatenar strings o usar caracteres de escape.
- **Expresiones switch mejoradas:** Se mejora aún más la funcionalidad introducida en Java 12.

3.3.3 Java 14

- **Records (en vista previa):** Nuevo tipo de clase que reduce la verbosidad al declarar clases que solo actúan como contenedores de datos inmutables.
- **Pattern Matching para instanceof:** Simplifica el uso del operador instanceof, eliminando la necesidad de realizar casting explícito.
- **Excepciones NullPointerException mejoradas:** Mejora la precisión de las trazas, indicando exactamente qué variable es null.

3.3.4 Java 15

- **Cambios internos y preliminares:** Muchos de los cambios en Java 15 son internos, con algunas características en vista previa o eliminadas.
- **Sealed Classes (vista previa):** Clases selladas que permiten restringir qué otras clases pueden heredar de ellas.

3.3.5 Java 16

- **Records:** Finalmente implementados de forma definitiva. Los records permiten declarar clases que actúan como simples contenedores de datos.
- **Pattern Matching para instanceof:** Finalización de esta característica que permite evitar casting explícito.
- **Herramienta de empaquetado (jpackage):** Permite empaquetar aplicaciones Java en instaladores nativos para diferentes plataformas (Windows, macOS, Linux).

3.3.6 Java 17

- **Versión LTS:** Java 17 es una versión de soporte a largo plazo (LTS) que sucedió a Java 11. Ofrece soporte hasta 2026, con un soporte extendido hasta 2029.
- **Encapsulación fuerte del JDK:** Se encapsulan firmemente las clases internas del JDK, mejorando la seguridad y el mantenimiento.
- **Clases sealed:** Implementación definitiva de las clases selladas, que limitan qué clases pueden extender una clase sellada.

3.4 Java 18 - 21

3.4.1 Java 18

- **UTF-8 por defecto:** Ahora la codificación de caracteres por defecto en la plataforma es UTF-8.
- **Servidor web simple:** Se introduce una utilidad en la línea de comandos para levantar un servidor web básico.
- **Javadoc mejorado:** Permite incluir fragmentos de código en los comentarios de la documentación.

3.4.2 Java 19

- **Virtual Threads (Vista previa):** Introduce threads virtuales que permiten aumentar drásticamente el número de hilos manejados por las aplicaciones.
- **Pattern Matching para switch:** Mejora el uso de switch para trabajar con patrones.
- **API de funciones y memoria externas:** Mejor integración con código nativo.

3.4.3 Java 20

No introduce novedades en el lenguaje, pero incluye mejoras en seguridad, rendimiento y correcciones.

3.4.4 Java 21

- **Virtual Threads:** Finalmente lanzados, permiten trabajar con millones de hilos de manera eficiente.
- **Sequenced Collections:** Introduce nuevas colecciones con un comportamiento más consistente.
- **Pattern Matching mejorado:** Extiende las capacidades del pattern matching, facilitando la implementación de lógica más robusta y concisa.

3.5 Metodos Streams

3.5.1 filter()

Este método devuelve un nuevo stream compuesto únicamente por los elementos que cumplen una condición determinada. Utiliza un Predicate para evaluar cada elemento del stream.

```
public Long getNumVuelosDia(Fecha f) {
    return vuelos.stream()
.filter(x -> x.getFecha().equals(f))
.count();
}
```

3.5.2 allMatch()

Este método devuelve true si todos los elementos del stream cumplen una condición determinada.

```
    public Boolean todosCompletos(Fecha f) {
    return vuelos.stream()
    .filter(x -> x.getFecha().equals(f))
    .allMatch(x -> x.getNumPasajeros().equals(x.getNumPlazas()));
}
```

3.5.3 anyMatch()

Este método devuelve true si al menos uno de los elementos del stream cumple una condición determinada.

```
    public Boolean hayVueloDestinoFecha(Fecha f, String d) {
    return vuelos.stream()
    .anyMatch(x -> x.getFecha().equals(f) && x.getDestino().equals(d));
    }
```

3.5.4 min()

Devuelve el elemento mínimo de un stream basado en un Comparator.

```
    public Vuelo getVueloMasBaratoDestino(String d) {
        return vuelos.stream()
          .filter(x -> x.getDestino().equals(d))
          .min(Comparator.comparing(Vuelo::getPrecio))
          .get();
      }
```

3.5.5 max()

Devuelve el elemento máximo de un stream basado en un Comparator.

```
public Vuelo getVueloMayorOcupacion(Fecha f) {
return vuelos.stream()
.filter(x -> x.getFecha().equals(f))
.max(Comparator.comparingDouble
(x -> x.getNumPasajeros() / x.getNumPlazas()))
.get();
}
```

3.5.6 map()

Este método transforma los elementos del stream a otro tipo mediante una función.

```
    public Double getMayorOcupacion(Fecha f) {
  return vuelos.stream()
    .filter(x -> x.getFecha().equals(f))
```

```
    .mapToDouble(x -> x.getNumPasajeros() / x.getNumPlazas())
    .max()
    .getAsDouble();
}
```

3.5.7 reduce()

Combina los elementos del stream en un solo valor utilizando una operación de acumulación.

```
    public Duracion getDuracionVuelosFecha(Fecha f) {
  return vuelos.stream()
    .filter(x -> x.getFecha().equals(f))
    .map(Vuelo::getDuracion)
    .reduce(new DuracionImpl(0, 0), Duracion::suma);
}
```

3.5.8 forEach()

Este método aplica una acción a cada elemento del stream.

```
    public void incrementaPrecios10pAPartirFecha(Fecha f) {
        vuelos.stream()
          .filter(x -> x.getFecha().compareTo(f) > 0)
          .forEach(x -> x.setPrecio(x.getPrecio() * 1.1));
      }
```

3.5.9 sorted()

Ordena los elementos del stream utilizando un Comparator.

```
    public void escribeVuelosOrdenadosFechaDuracion(String fileName) {
        Util.escribeFichero(vuelos.stream()
          .sorted(Comparator.comparing(Vuelo::getFecha)
          .thenComparing(Vuelo::getDuracion)), fileName);
      }
```

3.5.10 collect()

Este método transforma el stream en una colección o en otro tipo de estructura.

```
    public List<Duracion> getDuracionesDestino(String d) {
  return vuelos.stream()
    .filter(x -> x.getDestino().equals(d))
    .map(Vuelo::getDuracion)
    .collect(Collectors.toList());
}
```

4 Arquitectura de Software

La arquitectura de software se refiere al proceso de definir los componentes, relaciones y comportamientos que satisfacen los requisitos operacionales y técnicos de un sistema. El objetivo es cumplir con criterios como seguridad, eficiencia, disponibilidad y usabilidad.

4.1 Fundamentos de la arquitectura de software

Se abordan aspectos críticos que pueden afectar el éxito o fracaso del software, como el entorno de despliegue, la producción y el uso del sistema por parte de los usuarios. Se destaca la importancia de considerar los intereses de todos los agentes involucrados: los usuarios, el propio sistema y los objetivos del negocio. La arquitectura debe ser flexible para soportar cambios futuros tanto en software como hardware, y debe minimizar los riesgos asociados a su construcción.

4.2 Capacidades de la Arquitectura de Software

La arquitectura de software debe ser capaz de:

- Mostrar la estructura del software sin detallar la implementación.
- Abordar los casos de uso, los requisitos funcionales y de calidad.
- Permitir el cambio de software o hardware sin afectar la estructura del sistema.

4.3 Modelos de Arquitectura de Software

4.3.1 Arquitectura en Capas (N-Capas)

- Organiza la funcionalidad en capas, con roles claramente definidos. Cada capa puede residir en una misma máquina o distribuirse en varias.
- **Beneficios:** Aislamiento, modularidad, rendimiento mejorado y capacidad de prueba.
- **Cuándo usarla:** Cuando se necesitan varias capas reutilizables o cuando la lógica de negocio se expone a través de interfaces de servicio.

4.3.2 Arquitectura N-Niveles (N-Tier)

- Similar a la arquitectura en capas, pero con la separación de las capas en niveles físicos (servidores).
- **Beneficios:** Escalabilidad, mantenibilidad y disponibilidad, ya que cada nivel es independiente y puede redundarse.

4.4 Casos de Uso

Los casos de uso describen acciones que los usuarios o sistemas externos realizan en el software. Son representaciones de la funcionalidad pública de la aplicación.

4.5 Reglas de Negocio

Las reglas de negocio definen el comportamiento del sistema. Separar las reglas de negocio del resto de la aplicación facilita su evolución y asegura que los cambios en el sistema no afecten a otras partes.
Importancia: Las reglas de negocio son el núcleo de la arquitectura, alrededor del cual se construyen las otras capas.

4.6 Testing y Mantenimiento

Separar las reglas de negocio de la implementación facilita el mantenimiento y hace que la arquitectura sea "testable". Esta independencia también permite cambiar frameworks, bases de datos o bibliotecas sin alterar las reglas de negocio.

4.7 Patrones de Diseño

- La programación en capas organiza los objetos en tres capas principales: presentación, lógica de negocio y datos.
- **Capa de Presentación:** Interactúa con el usuario, mostrando y capturando datos.
- **Capa de Lógica de Negocio:** Contiene las reglas de negocio y procesa los datos.
- **Capa de Datos:** Gestiona la interacción con bases de datos y otros sistemas externos.

4.8 Ventajas y Desvantajas

- **Ventajas:** Modularidad, facilidad de mantenimiento y escalabilidad.
- **Desventajas:** La sobrecarga de capas puede reducir la eficiencia si no se balancea adecuadamente.

4.9 Ejemplo de Arquitectura en Capas

- N-Capas
- Modelo-Vista-Controlador (MVC)

5 Web Service

Un servicio web es una tecnología que utiliza protocolos y estándares para intercambiar datos entre aplicaciones desarrolladas en diferentes lenguajes de programación y plataformas, promoviendo la interoperabilidad.

5.1 SOAP

- SOAP es un protocolo que utiliza XML para el intercambio de información. Define cómo los servicios web se comunican independientemente de la plataforma o el lenguaje.
- SOAP permite la comunicación entre empresas y clientes mediante el uso de HTTP y XML para las solicitudes y respuestas.
- WSDL (Web Service Definition Language) se utiliza para describir los servicios disponibles en SOAP y cómo usarlos.

5.2 REST

- REST es un estilo de arquitectura que usa HTTP para el intercambio de información, simplificando las interacciones entre cliente y servidor.
- n REST, las URIs identifican los recursos y los verbos HTTP (GET, POST, PUT, DELETE) determinan las operaciones a realizar.

SOAP es más estructurado y permite un mayor control sobre la seguridad y transacciones, mientras que REST es más ligero y eficiente para aplicaciones web.

5.3 JSON

JSON (JavaScript Object Notation) es un formato de texto ligero para almacenar e intercambiar datos. Es fácil de leer y escribir para los humanos y simple para las máquinas de procesar.

5.3.1 Caracteristicas

- JSON usa comillas dobles para las claves y los valores, y su formato es similar a los objetos de JavaScript.
- Es más liviano y rápido de procesar que XML, lo que lo convierte en una alternativa popular para la transmisión de datos.

5.3.2 Tipos de valores

- Números: enteros o decimales.
- String: Una secuencia de caracteres Unicode.
- Booleanos: true o false.
- Arreglos: Una colección ordenada de valores.
- Objetos: Un conjunto de pares clave-valor..
- Nulo: valor nulo.

5.3.3 Ventajas

Es fácil de entender, rápido y más ligero que XML, además de ser nativamente soportado por JavaScript.

5.4 HTTP

HTTP (Hypertext Transfer Protocol) es un protocolo de comunicación utilizado en la web para transferir datos entre un cliente (generalmente un navegador web) y un servidor.

5.4.1 Funcionamiento

HTTP funciona mediante el intercambio de mensajes: el cliente envía solicitudes (requests) y el servidor devuelve respuestas (responses). Las solicitudes utilizan métodos (verbos) como GET, POST, PUT, DELETE, etc.

5.4.2 Versiones

- HTTP/1.1: Introdujo el uso de verbos como GET y POST.
- HTTP/2: Mejora el rendimiento mediante la compresión de cabeceras y el uso de una única conexión para múltiples solicitudes.
- HTTP/3: Introduce el uso de QUIC (un protocolo basado en UDP) para mejorar la velocidad y confiabilidad.

5.4.3 Codigos de respuesta

- 1xx: Respuestas informativas.
  - 100: Continuar.
  - 101: Cambio de protocolo.
  - 102: Procesando.
- 2xx: Respuestas satisfactorias.

  - 200: OK.
  - 201: Creado.
  - 202: Aceptado.
  - 203: Información no autoritativa.
  - 204: Sin contenido.
  - 205: Restablecer contenido.
  - 206: Contenido parcial.
- 3xx: Redirecciones.
  - 300: Múltiples opciones.
  - 301: Movido permanentemente.
  - 302: Encontrado.
- 4xx: Errores del cliente.
  - 400: Solicitud incorrecta.
  - 401: No autorizado.
  - 402: Pago requerido.
  - 403: Prohibido.
  - 404: No encontrado.
  - 405: Método no permitido.
  - 406: No aceptable.
  - 408: Tiempo de espera de solicitud.
- 5xx: Errores del servidor.
  - 500: Error interno del servidor.
  - 501: No implementado.
  - 502: Puerta de enlace incorrecta.
  - 503: Servicio no disponible.
  - 504: Tiempo de espera de puerta de enlace.
  - 505: Versión HTTP no compatible.

5.4.4 Cabeceras

Las cabeceras (headers) son metadatos que se envían junto con las solicitudes y respuestas HTTP.

5.4.5 Parametros

Se pueden agregar dos tipos de parámetros de PATH y de QUERY, la principal diferencia es que los de PATH son obligatorios ya forman parte del URL en si, mientras que los de QUERY son opcionales.

5.5 JWT

Un JWT es un estándar para transmitir de manera segura información entre dos partes como un usuario y un servidor. Contiene un conjunto de "claims" que son declaraciones sobre el usuario.

5.5.1 Estructura

Un JWT consta de tres partes separadas por puntos:

- Header: Contiene el tipo de token y el algoritmo de encriptación.
- Payload: Incluye los datos o claims, como la identidad del usuario.
- Signature: La firma se genera usando el header, payload y una clave secreta, y se utiliza para verificar la autenticidad del token.

5.5.2 Firma JWT

La firma asegura que el token no haya sido alterado. Si alguien modifica el token, la firma no coincidirá y se podrá rechazar.

5.5.3 Algortimos de encriptación

- HS256: Utiliza una clave secreta compartida entre el emisor y el receptor.
- RS256: Utiliza un par de claves pública y privada para la firma y validación.

6 Bases de datos

Un sistema de bases de datos es una colección de datos interrelacionados y un conjunto de programas que permiten a los usuarios acceder y modificar estos datos. Su propósito es ofrecer una visión abstracta de los datos, ocultando los detalles de cómo se almacenan.

6.1 Modelos de bases de datos

Los modelos de datos describen la estructura de las bases de datos, sus relaciones, y restricciones de consistencia en los niveles físico, lógico y de vistas.

6.2 Base de datos relacional

Se basan en tablas para representar datos y relaciones entre ellos. El lenguaje SQL se utiliza comúnmente para manipular estas bases de datos.

6.3 Restricciones de Integridad

Las restricciones de integridad aseguran que las modificaciones a la base de datos no comprometan su consistencia. Esto incluye la integridad referencial, que garantiza que las claves externas se correspondan con valores válidos en otras tablas.

6.4 SQL

SQL permite definir, manipular y consultar bases de datos. Las principales operaciones incluyen la creación de tablas, la inserción de datos, la consulta y la actualización de registros.

6.5 Normalización

La normalización es un proceso para estructurar las bases de datos, eliminando redundancias y asegurando la consistencia de los datos. Se utilizan formas normales para guiar este proceso, ayudando a organizar los datos de manera eficiente.

6.5.1 Reglas de normalización

- **Primera forma normal (1FN):** Elimina la repetición de datos. Cada valor en una columna debe ser atómico, es decir, no debe contener múltiples valores. Ejemplo: en lugar de repetir los datos de un cliente en cada venta, se crea una tabla separada de clientes, referenciada por una clave.
- **Segunda forma normal (2FN):** Asegura que todas las columnas de una tabla dependan únicamente de la clave primaria. Si algunas columnas no dependen de la clave principal, deben separarse en nuevas tablas. Ejemplo:

crear una tabla separada para detalles repetidos, como fechas de ventas o productos vendidos.

- **Tercera forma normal (3FN):** Elimina dependencias transitivas, donde una columna no clave depende de otra columna no clave. Ejemplo: los detalles del proveedor de un producto deben estar en una tabla separada si no dependen de la clave principal de la tabla de ventas.

La normalización mejora el diseño de las bases de datos al reducir la redundancia y mejorar la consistencia de los datos, haciendo que las bases sean más fáciles de mantener y ampliar.