Presentado por:

Juan David Garcia, Karen, Santiago, Marlon Torres, jhon corredor

corre

JSON WEB TOKEN

¿Qué es?

Un JWT (JSON Web Token) es un estándar abierto (RFC 7519) que define un formato compacto y autónomo para transmitir información de forma segura entre dos partes como un objeto JSON. Este token es típicamente utilizado para la autenticación y autorización en aplicaciones web y servicios API.

Los JWT son utilizados comúnmente para autenticación y autorización en aplicaciones web y servicios en línea. Son populares porque son compactos, fáciles de transmitir y pueden ser verificados fácilmente por el receptor sin necesidad de consultar una base de datos centralizada.

Estructura

-Header (Encabezado): contiene información sobre el tipo de token y el algoritmo de firma utilizado.
-Payload (Cuerpo): lleva la carga útil del token, es decir, la información que se desea transmitir.
-Signature (Firma): se utiliza para verificar que el token no ha sido alterado durante la transmisión y que proviene de una fuente confiable.

Ejemplo

corre

```
// Encabezado (Header)
{
  "alg": "HS256", // Algoritmo de firma utilizado (en este caso, HMAC SHA-256)
  "typ": "JWT"    // Tipo de token
}
// Cuerpo (Payload)
{
  "usuario_id": 123456789,      // ID del usuario
  "nombre_usuario": "ejemplo", // Nombre del usuario
  "rol": "admin",                // Rol del usuario (en este caso, administrador)
  "exp": 1632345600              // Tiempo de expiración del token (en este caso, 23 de septiembre de 2021)
}
// Firma (Signature)
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

Ejemplo:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMjM0NTY3ODkwLCJuYW1lIjoiZXN0ZXJlbyIsInJvbCI6ImFkbWluIiwiZXhwIjoxNjMyMzQ1NjAwfQ.ji2M9eapc9G0hxrFSSMQS3r5B5bqQLXzPWl60eSkZ6o

Mensajeria en un solo lugar

Estructura

1. Definir un módulo de mensajes.
2. Registrar todos los mensajes en el módulo.
3. Usar diferentes niveles de severidad.
4. Filtrar y visualizar los mensajes.
5. Configurar el destino de los mensajes.

Implementación de un módulo de registro:

- Crear un módulo o clase dedicado a la gestión de mensajes.
- Implementar funciones para registrar diferentes tipos de mensajes (errores, advertencias, información, etc.).
- Definir un formato estándar para los mensajes, incluyendo la fecha, el tipo de mensaje, el origen y el contenido.
- Centralizar la lógica de impresión o visualización de los mensajes en el módulo de registro.

2. Uso de un archivo de registro:

- Almacenar los mensajes en un archivo de texto o una base de datos.
- Configurar el nivel de detalle de los mensajes que se registran.
- Implementar un sistema de rotación de archivos para evitar que el archivo de registro crezca demasiado.
- Utilizar herramientas para analizar y filtrar los mensajes del archivo de registro.

```java
import java.util.logging.Logger;

public class Mensajes {

    private static final Logger logger = Logger.getLogger(Mensajes.class.ge

    public static void error(String mensaje) {
        logger.severe(mensaje);
    }

    public static void advertencia(String mensaje) {
        logger.warning(mensaje);
    }

    public static void info(String mensaje) {
        logger.info(mensaje);
    }

    public static void main(String[] args) {
        Mensajes.error("Ha ocurrido un error");
        Mensajes.advertencia("Se ha detectado un problema potencial");
        Mensajes.info("El programa se ha iniciado correctamente");
    }
}
```

Implementación de Cadenas Genéricas

Las cadenas genéricas son una herramienta poderosa en lenguajes de programación que permiten trabajar con tipos de datos sin necesidad de conocerlos de antemano.
Esto ofrece mayor flexibilidad y reusabilidad al código.

mplementar:

Parámetros de tipo: Se utilizan para especificar el tipo de dato que se va a almacenar en la cadena.
Clases genéricas: Se utilizan para crear clases que pueden trabajar con diferentes tipos de datos.
Funciones genéricas: Se utilizan para crear funciones que pueden trabajar con diferentes tipos de datos.

Ejemplo

Java

```java
public class MiClase<T> {

  private List<T> lista;

  public MiClase() {
    this.lista = new ArrayList<>();
  }

  public void agregar(T elemento) {
    this.lista.add(elemento);
  }

  public T obtener(int indice) {
    return this.lista.get(indice);
  }

}

MiClase<String> miClase = new MiClase<>();
miClase.agregar("Hola");
miClase.agregar("Mundo");
String valor = miClase.obtener(0);
System.out.println(valor);  // Salida: Hola
```

santi

Thank you!

Do you have any questions for me before we go?

karen

Que es @Query

La anotación `@Query` en Spring Data JPA te permite definir y ejecutar consultas personalizadas en repositorios JPA. Puedes escribir consultas JPQL o SQL nativas según tus necesidades. Esta anotación ofrece flexibilidad para manejar consultas más complejas que no pueden ser expresadas fácilmente con los métodos estándar del repositorio JPA. Al agregar `@Query` a un método en el repositorio JPA, puedes proporcionar la consulta deseada y especificar si es JPQL o SQL nativa. En resumen, `@Query` te brinda mayor control sobre cómo interactúas con la base de datos en tu aplicación Java.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class ProductoService {

    @Autowired
    private ProductoRepository productoRepository;

    public List<Producto> buscarPorNombre(String nombre) {
        return productoRepository.findByNombreContaining(nombre);
    }
}
```

karen

como aplicar sql nativo en el repositorio jpa

En una aplicación Java con JPA, definimos entidades para representar las tablas de la base de datos. Los repositorios JPA nos ayudan a interactuar con estas entidades, ofreciendo métodos para guardar, buscar, actualizar y eliminar datos.
Cuando necesitamos consultas más complejas que no pueden expresarse fácilmente con métodos estándar del repositorio JPA, podemos usar consultas SQL nativas. Para hacerlo:

1. **Escribimos la consulta SQL deseada, con todas las características de SQL que necesitemos.**
2. **Agregamos la anotación @Query a un método en el repositorio JPA, indicando que es una consulta SQL nativa.**
3. **Llamamos a este método en nuestro código para ejecutar la consulta SQL nativa y obtener los resultados.**

Ejemplo

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;
import java.util.List;

@Repository
public interface ProductoRepository extends JpaRepository<Producto, Long> {

    @Query(value = "SELECT * FROM productos WHERE precio > :valor", nativeQuery = true)
    List<Producto> findByPrecioMayorQue(double valor);
}
```

Supongamos que tenemos una entidad Producto que representa la tabla productos en nuestra base de datos. Queremos buscar todos los productos cuyo precio sea mayor que cierto valor. Podemos escribir la consulta SQL nativa correspondiente y agregarla a nuestro repositorio JPA:

Explicacion

karen

- **@Query indica que estamos proporcionando una consulta SQL nativa.**
- **value contiene la consulta SQL nativa que queremos ejecutar.**
- **nativeQuery = true le dice a Spring Data JPA que esta consulta es nativa.**

Luego, en nuestro servicio o controlador, podemos llamar al método findByPrecioMayorQue para ejecutar la consulta SQL nativa y obtener los resultados.