CryptoUtils - Utilidades Criptográficas

Módulo de utilidades criptográficas para Smart Parking que proporciona funciones seguras de hashing de contraseñas, generación de tokens y validaciones, utilizando únicamente APIs nativas de Node.js sin dependencias externas.

🔐 Características de Seguridad

- **Hashing de contraseñas** con algoritmo scrypt y salt único
- **Tokens HMAC** personalizados para autenticación
- **Comparación de tiempo constante** contra timing attacks
- **Rate limiting** para prevenir ataques de fuerza bruta
- **Sin dependencias externas** - solo APIs nativas de Node.js

📦 Funciones Principales

🔑 Manejo de Contraseñas

`hashPassword(plainPassword: string): string`

Genera un hash seguro de la contraseña usando scrypt con salt único.

```typescript
import { hashPassword } from './crypto-utils';

const hashedPassword = hashPassword('miContraseñaSegura123!');
// Retorna: "a1b2c3d4...hex_salt:e5f6g7h8...hex_hash"
```

Características:

- Salt de 32 bytes generado aleatoriamente
- Scrypt con parámetros seguros (N=16384, r=8, p=1)
- Hash de 64 bytes de longitud
- Formato: `salt:hash` en hexadecimal

`verifyPassword(plainPassword: string, hashedPassword: string): boolean`

Verifica si una contraseña en texto plano coincide con el hash almacenado.

```typescript
import { verifyPassword } from './crypto-utils';

const isValid = verifyPassword('miContraseñaSegura123!', storedHash);
// Retorna: true si coincide, false si no
```

Características:

- Comparación de tiempo constante (`timingSafeEqual`)
- Validación de formato del hash
- Manejo seguro de errores

🎫 Manejo de Tokens

`generateSimpleToken(payload: any, secret?: string): string`

Genera un token HMAC personalizado para autenticación.

```typescript
import { generateSimpleToken } from './crypto-utils';

const token = generateSimpleToken({
  adminId: 'admin-123',
  tenantId: 'tenant-uuid',
  email: 'admin@example.com'
});
// Retorna: "eyJwYXlsb2FkIjp7Li4ufQ.a1b2c3d4e5f6..."
```

Estructura del token:

- Timestamp para expiración
- Nonce aleatorio para unicidad
- Payload codificado en base64url
- Firma HMAC SHA-256

`validateSimpleToken(token: string, secret?: string, maxAge?: number): any`

Valida un token y retorna el payload si es válido.

```typescript
import { validateSimpleToken } from './crypto-utils';

const payload = validateSimpleToken(token, secret, 24 * 60 * 60 * 1000); // 24 horas
// Retorna: payload original o null si es inválido/expirado
```

`generateRefreshToken(payload: any): string`

Genera un refresh token con secret diferente para mayor seguridad.

```typescript
import { generateRefreshToken } from './crypto-utils';

const refreshToken = generateRefreshToken({
  adminId: 'admin-123',
  type: 'refresh'
});
```

🆔 Generación de IDs

`generateSecureId(): string`

Genera IDs únicos y seguros.

```typescript
import { generateSecureId } from './crypto-utils';

const sessionId = generateSecureId();
// Retorna: "1a2b3c4d-e5f6g7h8i9j0k1l2"
```

🛡️ Rate Limiting

`checkRateLimit(identifier: string, maxAttempts?: number, windowMs?: number): boolean`

Implementa rate limiting simple para prevenir ataques de fuerza bruta.

```typescript
import { checkRateLimit } from './crypto-utils';

const canAttempt = checkRateLimit('user@example.com', 5, 15 * 60 * 1000);
if (!canAttempt) {
  throw new Error('Too many login attempts');
}
```

Parámetros por defecto:

- `maxAttempts`: 5 intentos
- `windowMs`: 15 minutos (900,000 ms)

🏗️ Uso en el Proyecto

1. Creación de Administradores (Seed)

```typescript
// En seed.ts
const adminPassword = hashPassword('admin123');
await prisma.administrator.create({
  data: {
    email: 'admin@example.com',
    passwordHash: adminPassword,
    // ...otros campos
  }
});
```

2. Login de Administradores

```typescript
// En AdminLoginUseCase
export class AdminLoginUseCase {
  async execute(input: AdminLoginInput): Promise<AdminAuthOutput> {
    const admin = await this.adminRepository.findByEmail(input.email);

    // Verificar contraseña
    if (!admin.verifyPassword(input.password)) {
      throw new Error('Invalid credentials');
    }

    // Generar tokens
    const accessToken = generateSimpleToken(payload);
    const refreshToken = generateRefreshToken(payload);

    return { admin, authentication: { access_token: accessToken, ... } };
  }
}
```

3. Validación de Tokens

```typescript
// En AdminController
@Post('validate-token')
async validateToken(@Body() body: { token: string }) {
  const payload = validateSimpleToken(body.token);

  if (!payload) {
    return { valid: false };
  }

  return { valid: true, admin: payload };
}
```

⚙️ Configuración

Variables de Entorno

```bash
# Token secrets (requeridos en producción)
TOKEN_SECRET=tu-secret-super-seguro-aqui
REFRESH_SECRET=tu-refresh-secret-diferente-aqui
```

Parámetros de Scrypt

```typescript
const SCRYPT_OPTIONS = {
  N: 16384,        // Factor de costo (2^14)
  r: 8,            // Tamaño de bloque
  p: 1,            // Paralelización
  maxmem: 64 * 1024 * 1024 // 64MB máximo de memoria
};
```

🔒 Consideraciones de Seguridad

✅ Buenas Prácticas Implementadas

- **Nunca almacenar contraseñas en texto plano**
- **Salt único por contraseña** (32 bytes aleatorios)
- **Algoritmo scrypt** resistente a ataques con hardware especializado
- **Comparación de tiempo constante** para prevenir timing attacks
- **Tokens con expiración** y nonce único
- **HMAC para integridad** de tokens
- **Rate limiting** para prevenir fuerza bruta

⚠️ Recomendaciones Adicionales

- Usar secrets diferentes para tokens y refresh tokens
- Implementar rotación de secrets en producción
- Monitorear intentos de login fallidos
- Considerar 2FA para mayor seguridad
- Validar longitud y complejidad de contraseñas

🧪 Ejemplo de Uso Completo

typescript

```typescript
import {
  hashPassword,
  verifyPassword,
  generateSimpleToken,
  validateSimpleToken,
  checkRateLimit
} from './crypto-utils';

// 1. Registrar admin con contraseña encriptada
const passwordHash = hashPassword('MiContraseña123!');

// 2. Verificar login con rate limiting
const canTryLogin = checkRateLimit(email, 5, 15 * 60 * 1000);
if (!canTryLogin) {
  throw new Error('Too many attempts');
}

const isValidPassword = verifyPassword(plainPassword, passwordHash);
if (!isValidPassword) {
  throw new Error('Invalid credentials');
}

// 3. Generar token de acceso
const token = generateSimpleToken({
  adminId: admin.id,
  tenantId: admin.tenantId,
  email: admin.email
});

// 4. Validar token en requests posteriores
const payload = validateSimpleToken(token);
if (!payload) {
  throw new Error('Invalid or expired token');
}
```

📝 Notas Técnicas

- **Sin dependencias externas**: Solo usa APIs nativas de Node.js
- **Rendimiento**: Scrypt es computacionalmente intensivo por diseño
- **Memoria**: Configurado para usar máximo 64MB por operación
- **Compatibilidad**: Node.js 16+ requerido para `timingSafeEqual`

🔗 Enlaces Relacionados

- Node.js Crypto Documentation
- OWASP Password Storage Cheat Sheet
- RFC 7914 - The scrypt Password-Based Key Derivation Function