Contenido

1 Redis

Redis es **un proceso que corre en un servidor**, y su papel es actuar como **un almacén de datos muy rápido en memoria**. Lo describo en detalle:

1.1 Qué es Redis

Redis = **REmote DIctionary Server**. Es básicamente un **diccionario gigante en RAM** formado por parejas {clave → valore}.

Ejemplo:

> *sess:abc123 → { "userId": "u1", "roles": ["user"], "iat": 1731345000000 }*

A redis se le puede pedir, guardar, borrar o actualizar datos en milésimas de segundo.

1.2 Qué hace Redis en el proyecto SkyPort

Redis guarda las **sesiones activas**, es decir, cada vez que un usuario inicia sesión:

1. **Express** crea una cookie en su navegador (skyport.sid = abc123).
2. **Redis** guarda el contenido real de esa sesión bajo la clave sess:abc123.
3. En cada petición, el servidor mira la cookie, busca en Redis y sabe "ah, este SID pertenece al usuario X".

Así que Redis es **el lugar donde residen los datos de sesión** (en lugar de estar solo en memoria del proceso Node).

1.3 Qué ventajas proporciona utilizar Redis

- **Persistencia entre reinicios del servidor**: si nuestra app se reinicia, las sesiones siguen ahí.
- **Escalabilidad**: si existen varios procesos o servidores, todos pueden compartir las mismas sesiones porque leen el mismo Redis.
- **Velocidad**: todo está en memoria, por eso es instantáneo.
- **Control**: se puede destruir una sesión en cualquier momento borrando su clave (DEL sess:abc123).

1.4 Qué previene o evita Redis

Sin Redis, Express usaría **MemoryStore** → cada proceso tendría sus sesiones por separado. Eso genera problemas:

- si una app se reinicia, se pierden todas las sesiones;
- si se usan varios servidores, cada uno tiene las suyas y un usuario puede parecer "deslogueado" a medias;
- más propenso a fugas de memoria.

Redis **centraliza** todo eso y evita inconsistencias o pérdidas. 🖥️.

Nota: Memurai es la versión para Windows de Redis, es simplemente **el servicio Redis corriendo**. Se puede ver en los servicios de Windows como:

> ***Nombre**: Memurai*
> ***Estado**: Running*
> ***Puerto**: 6379*

La aplicación **Node se conecta a Redis (como cliente)** mediante:

const redisClient = createClient({ url: 'redis://localhost:6379' });

En resumen: Redis es un **proceso auxiliar** que mantiene datos efímeros (como sesiones, caché o colas) en memoria, para que tu servidor web sea más rápido, más estable y escalable.

2 Instalación de Memurai (Redis para Windows)

La instalación es bastante directa.

2.1 Requisitos previos

- **Windows de 64 bits**, idealmente Windows 10 o Windows Server 2016+ (o superior) según la documentación oficial. Redis+1
- **Tener privilegios de administrador** para instalar servicios en Windows (si se decide instalarlo como servicio).

~~2.2 Instalación manual (GUI)~~

~~1. Visitar la página de descarga: "Get Memurai" en la web oficial. memurai.com+2memurai.com+2~~
~~2. Descarga la edición **Developer** (gratuita para desarrollo/pruebas) o la **Enterprise** si es para producción. memurai.com+1~~
~~3. Ejecuta el archivo .msi.~~
~~4. Durante el instalador puedes:~~
~~- Elegir instalar como **servicio de Windows** (poner que arranque al inicio)~~
~~- Especificar el puerto (por defecto 6379)~~
~~- Crear una regla de firewall si lo deseas Redis+1~~
~~5. Una vez instalado, verifica que el servicio esté corriendo desde el "Administrador de servicios" de Windows, o usa en PowerShell algo como ***net start memurai*** si se instaló como servicio.~~
~~6. Usa el cliente memurai-cli.exe (viene con la instalación) para probar, por ejemplo:~~
~~7. memurai-cli.exe ping~~

~~Deberías obtener PONG.~~

2.3 Instalación por línea de comandos / silenciosa

Mediante ""**winget**": **Desde PowerShell como administradores:**

PS C:\Windows\System32> winget install -e --id Memurai.MemuraiDeveloper

Encontrado Memurai Developer [Memurai.MemuraiDeveloper] Versión 4.1.2
El propietario de esta aplicación le concede una licencia.
Microsoft no es responsable, ni tampoco concede ninguna licencia de paquetes de terceros.
Descargando https://dist.memurai.com/releases/Memurai-Developer/4.1.2/Memurai-Developer-v4.1.2.msi
[redacted] *8.35 MB / 8.35 MB*
El hash del instalador se verificó correctamente
Iniciando instalación de paquete...
Instalado correctamente

Se añade el **path** donde está Memurai ejecutando el siguiente comando:

[Environment]::SetEnvironmentVariable("Path", $env:Path + ";C:\Program Files\Memurai", "Machine")

Comprobamos si todo está instalado correctamente

memurai-cli.exe ping

Si todo ha ido bien, debería contestar:

PONG

2.4 Configuración adicional

El archivo de configuración está en ***memurai.conf*** (en la carpeta de instalación). Se puede cambiar: puerto, persistencia, memoria, etc. Redis+1

Si memurai se instaló como servicio y decidimos cambiar algo en memurai.conf, luego deberá **reiniciarse** el servicio para que los cambios surtan efecto.

Deberá verificarse que no haya otro servicio usando el puerto 6379 (o el que hayamos elegido) para evitar conflicto.

2.5 Verificar que todo funciona

Desde PowerShell o CMD:

memurai-cli.exe ping

Debería obtenerse

PONG.

En nuestra aplicación web (por ejemplo el backend Node.js/Express que usa Redis/Knex u otro caché), apunta a localhost:6379 (o el puerto que hayas configurado) y verifica que puede conectarse.

2.6 Arrancar / parar el servicio

Parar:

Stop-Service Memurai

Arrancar:

Start-Service Memurai

Con los comandos “viejos”:

net stop memurai

net start memurai

3 Usarlo en app Node (SkyPort, sesiones, etc.)

3.1 En el archivo “.env”

REDIS_URL=redis://localhost:6379

3.2 Activación de sesiones

En "***middleware/session.js"*** tenemos esto:

```
const session = require('express-session');
const { createClient } = require('redis');
const RedisStore = require('connect-redis').default;
const { DEV, timeouts, cookieOptions, redisConfig } = require('../config/sessions');

const redisClient = createClient({ url: redisConfig.url });
redisClient.on('error', (err) => console.error('[Redis] Error', err));
redisClient.connect().catch(err => console.error('[Redis] No conecta:', err));

module.exports = session({
  name: cookieOptions().name,
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
  rolling: true,
  cookie: cookieOptions().cookie,
  store: new RedisStore({
    client: redisClient,
    prefix: redisConfig.prefix,
    ttl: Math.floor(timeouts.idleMs / 1000)
  })
});
```

Y en "***app.js"*** tenemos:

```
app.use(require('./middleware/session'));
```

👉 A partir de esa línea en *app.js*, todas las peticiones que pasen por ahí tendrán **req.session disponible**, y se gestionará la cookie de sesión + Redis.

3.3 ¿Cuándo se "crea" la sesión vacía?

En la creación de la sesión hay dos momentos clave:

3.3.1 Creación implícita por express-session

Cuando llega una **petición sin cookie de sesión previa**: El middleware de express-session:

- Genera un **SID** nuevo (identificador de sesión).
- Crea un objeto **req.session** para esa petición.

Pero como tenemos por código **saveUninitialized: false**, NO guarda nada en Redis ni envía cookie al navegador hasta que modifiques la sesión.

Es decir: a nivel de código ya se puede hacer **req.session.algo**, pero hasta que no toques la sesión, no se persiste.

👉 Aquí la "creación" es automática, pero invisible para nosotros salvo que usemos **req.session**.

3.3.2 Creación "limpia" y explícita en el login / registro

En el "**login"** y en el "**registro con autoLogin"** utilizamos esto:

```
await new Promise(resolve => req.session.regenerate(resolve));
```

Este **sí es el gran momento oficial de "crear sesión vacía"** para un usuario autenticado.

3.3.3 En /api/auth/login

```
// --- /api/auth/login ---
router.post('/login', rateLogin, async (req, res) => {
  // ...
  const ok = await bcrypt.compare(pass, candidate.passwordHash);
  if (!ok) return res.status(401).json({ error: 'credenciales' });

  clearLoginRateForIp(req.ip);

  // 🔴 AQUÍ se crea una SESIÓN NUEVA y VACÍA
  await new Promise(resolve => req.session.regenerate(resolve));

  // 🔵 AQUÍ SE EMPIEZA A RELLENAR
  req.session.userId = candidate.id;
  req.session.roles  = candidate.roles || [];
  req.session.iat    = Date.now();

  // 🟢 AQUÍ SE GUARDA EN REDIS Y SE ENVÍA LA COOKIE AL NAVEGADOR
  req.session.save(err => {
    if (err) return res.status(500).json({ error: 'session_save' });
    res.json({ ok: true, userId: candidate.id, roles: candidate.roles || [] });
  });
});
```

- **req.session.regenerate(...)**:
  - Destruye la sesión previa (si había).
  - Crea un **nuevo SID**.
  - Crea un nuevo req.session **vacío** asociado a ese nuevo SID.
- Después de eso, la sesión está vacía hasta que se le meten cosas.

3.4 ¿Cuándo y dónde se rellena la sesión?

La sesión se rellena básicamente en **login** y en **register con autoLogin**.

3.4.1 En el login

```
req.session.userId = candidate.id;       // ID del usuario logueado
req.session.roles  = candidate.roles || []; // roles associated (admin, player...)
req.session.iat    = Date.now();         // "issued at" (marca de tiempo)
```

Después de estas asignaciones, la sesión ya no está vacía → tiene todos los datos que luego utilizarán los middlewares de auth.

3.4.2 En el registro (si autoLogin !== false)

```
// --- /api/auth/register ---
if (autoLogin !== false) {
  await new Promise(resolve => req.session.regenerate(resolve));  // SESIÓN NUEVA Y VACÍA
  req.session.userId = newUser.id;       // RELLENAS
```

```
    req.session.roles  = newUser.roles || []; // RELLENAS
    req.session.iat    = Date.now();        // RELLENAS

    return req.session.save(err => {        // GUARDA y ENVÍA COOKIE
      if (err) return res.status(500).json({ error: 'session_save' });
      res.status(201).json({ ok: true, userId: newUser.id, roles: newUser.roles });
    });
  }
```

Patrón idéntico al login:

1. regenerate → **sesión vacía nueva**
2. asignaciones → **meter contenido**
3. req.session.save → guardar en Redis + mandar cookie

3.5 ¿Cuándo y dónde se borra la sesión?

3.5.1 Logout explícito

En /api/auth/logout:

```
router.post('/logout', requireAuth, (req, res) => {
  req.session.destroy(err => {        // 🔴 BORRA SESIÓN EN REDIS
    if (err) return res.status(500).json({ error: 'session_destroy' });

    const opts = cookieOptions().cookie;

    // 🔴 BORRA LA COOKIE EN EL NAVEGADOR
    res.clearCookie(COOKIE_NAME, {
      path: opts.path,
      sameSite: opts.sameSite,
      secure: opts.secure,
      httpOnly: true
    });

    return res.json({ ok: true });
  });
});
```

Aquí pasan dos cosas:

1. **req.session.destroy(...)**
   - Le dice al RedisStore que borre la clave sess:<sid> de Redis.
   - A partir de ese momento, aunque el navegador envíe esa cookie, ya no habrá datos de sesión asociados a ese SID.
2. **res.clearCookie(COOKIE_NAME, ...)**
   - Envía una cabecera Set-Cookie que indica al navegador que **borre la cookie** skyport.sid (o el nombre que tengas).
   - Funde el vínculo entre el navegador y el SID.

Resultado: **logout completo**: ni navegador ni servidor recuerdan esa sesión.

3.5.2 Borrado automático por inactividad

En config/sessions.js:

```
const idleMin = parseInt(process.env.SESSION_IDLE_MIN || '30', 10);
const timeouts = {
  idleMs: idleMin * 60 * 1000,
  absoluteMs: absoluteHrs * 60 * 60 * 1000
};
```

Y en middleware/session.js:

```
store: new RedisStore({
  client: redisClient,
  prefix: redisConfig.prefix,
  ttl: Math.floor(timeouts.idleMs / 1000) // TTL en segundos
})
```

Además, la cookie:

```
cookie: {
 // ...
 maxAge: timeouts.idleMs
}
```

Esto significa:

- **En Redis**: la sesión tiene un TTL = idleMs. Si el usuario está idle más tiempo del TTL, Redis borra la sesión.
- **En el navegador**: la cookie caduca (maxAge) tras ese mismo tiempo de inactividad.

Y como además tenemos rolling: true:

rolling: true, // renueva cookie + TTL en cada petición

- Cada vez que el usuario hace una petición, se renuevan:
    - El maxAge de la cookie
    - El TTL de la sesión en Redis

Si el usuario deja de hacer peticiones, tras idleMs:

- La sesión **expira en Redis**.
- La cookie caduca en el navegador.

Eso es un **borrado automático por inactividad** (no “logout” manual, pero efecto muy similar).