7. Revoking Tokens

Sometimes you must **invalidate** a token before it expires — for example, after detecting suspicious activity or user logout.

Common revocation strategies:

- **Blacklist invalid tokens** in a database or cache (e.g., Redis).
- **Rotate secrets** periodically — invalidates all old tokens.
- **Track token versions** per user and reject outdated ones.

Example using a Redis blacklist:

```
import redis from "redis";
const client = redis.createClient();

async function revokeToken(token) {
  const decoded = jwt.decode(token);
  await client.set(`revoked:${decoded.jti}`, "true", "EX",
decoded.exp - Math.floor(Date.now() / 1000));
}

async function isTokenRevoked(token) {
  const decoded = jwt.decode(token);
  return client.get(`revoked:${decoded.jti}`);
}
```

8. Handling Expiration and Grace Periods

Tokens should **fail fast** after expiration to prevent reuse.

However, in high-latency or distributed systems, a **grace period** (a few seconds) can prevent false negatives.

Example:

```
const now = Math.floor(Date.now() / 1000);
if (decoded.exp < now - 5) return res.status(403).json({
error: "Token expired" });
```

9. API Keys for Service-to-Service Authentication

While JWTs are great for user-level authentication, simple **API keys** are often sufficient for **trusted internal tools**.

Pattern:

- Each service gets its own key.

- Keys are rotated periodically.
- Keys are stored as environment variables.

Example middleware:

```
function apiKeyAuth(req, res, next) {
  const key = req.headers["x-api-key"];
  if (key !== process.env.INTERNAL_API_KEY)
    return res.status(403).json({ error: "Invalid API key"
});
  next();
}
```

10. Secure Token Storage

For clients:

- Never store tokens in plaintext or unencrypted local storage.
- Use `SecureStore` or encrypted files on mobile and desktop apps.

For servers:

- Store refresh tokens hashed in databases.
- Log only token fingerprints, never full tokens.

Example:

```
import crypto from "crypto";
const fingerprint =
crypto.createHash("sha256").update(token).digest("hex");
```

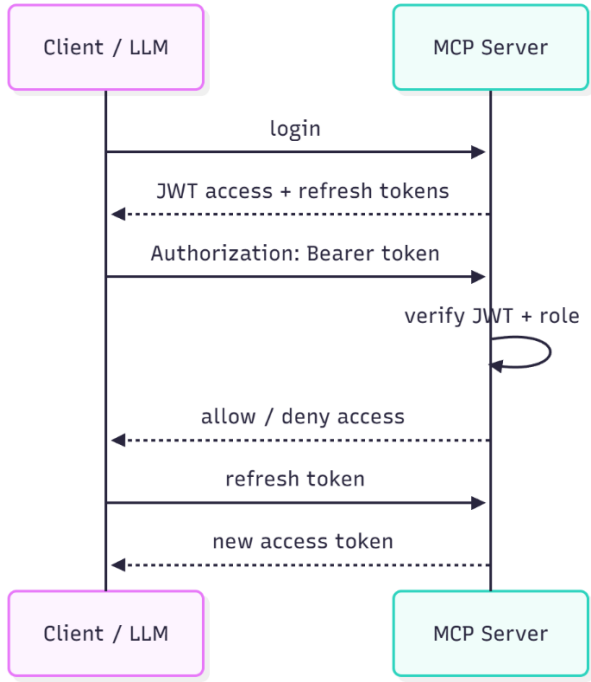
11. Token Rotation and Zero-Trust

For mission-critical MCP systems:

- Use **short-lived access tokens** (5–15 minutes).
- Enforce **continuous re-authentication**.
- Rotate secrets regularly.
- Treat all incoming connections as untrusted until verified.

This follows the **Zero-Trust Security Model**, where no token or session is assumed safe until proven valid every time.

12. Example MCP Auth Flow Diagram



13. Common Mistakes to Avoid

Mistake	Risk	Solution
Tokens with no expiration	Unlimited access if leaked	Always set `exp` claims
Storing tokens in localStorage	XSS exposure	Use secure cookies or memory storage
Hardcoded secrets in code	Key exposure	Store secrets in environment or vault
No token rotation	Stale credentials persist	Rotate regularly
Ignoring revocation	Tokens remain valid after logout	Implement blacklist or secret rotation

Securely Connecting Clients and Servers

Secure communication between MCP clients and servers is essential — not optional. Because MCP servers often act as intermediaries between AI models, APIs, and user systems, even a minor security gap can expose sensitive data or credentials.

This section explains **how to establish, maintain, and enforce secure channels** for communication across all environments.

1. Why Secure Connections Matter

Every interaction between an MCP client and server involves data exchange — sometimes user credentials, API keys, or proprietary context.

If not encrypted, these can be intercepted by attackers through:

- **Man-in-the-Middle (MITM)** attacks
- **Spoofed endpoints**
- **Unauthorized eavesdropping**

Using **secure connections (HTTPS, TLS, and authentication tokens)** ensures that:

- Data remains **private (confidentiality)**
- Messages cannot be altered (integrity)
- Only verified entities can communicate (authenticity)

2. Using HTTPS and TLS

The first and most fundamental step in securing MCP connections is enforcing

HTTPS (TLS 1.2 or higher).

Steps to enable HTTPS:

1. Obtain an SSL/TLS certificate
2. Use **Let's Encrypt** for free automated certificates.
3. Or buy one from a trusted CA for production.
4. Configure your server to listen on port **443** with TLS enabled.
5. Redirect all HTTP traffic to HTTPS automatically.

Example (Node.js + Express + HTTPS):

```
import https from "https";
import fs from "fs";
import app from "./app.js";

const options = {
  key: fs.readFileSync("server.key"),
  cert: fs.readFileSync("server.crt"),
};

https.createServer(options, app).listen(443, () => {
  console.log("☑ MCP Server running securely on
https://localhost:443");
});
```

Best Practice:

Never transmit any authentication token, API key, or user data over plain HTTP.

3. Verifying Client Identity

Beyond encryption, your MCP server should verify WHO is connecting. This can be done through:

Method	Description	Example Use Case
API Keys	Static shared secrets	Internal service-to-service authentication
JWT Tokens	Signed, time-limited tokens	User sessions or tool integrations
Mutual TLS (mTLS)	Both client and server use certificates	High-security or enterprise setups

Mutual TLS Example:

Both the client and server present certificates signed by a trusted CA.

```
# On server side
openssl req -new -x509 -key server.key -out server.crt
# On client side
openssl req -new -x509 -key client.key -out client.crt
```

Server configuration example:

```
const server = https.createServer({
  key: fs.readFileSync("server.key"),
  cert: fs.readFileSync("server.crt"),
  ca: fs.readFileSync("ca.crt"),
  requestCert: true,
  rejectUnauthorized: true,
}, app);
```

This ensures that **only authorized clients** can communicate with the MCP server.

4. Implementing Token-Based Authentication

Combine secure transport (TLS) with **application-level tokens** for layered protection.

Each client or LLM agent should have:

- A unique **access token**
- Optional **scopes or permissions**
- Token **expiration times** to reduce misuse

Request headers example:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6...
```

Then validate the token in every incoming request:

```
if (!verifyToken(req.headers.authorization)) {
```

```
  return res.status(401).json({ error: "Unauthorized" });
}
```

5. Encrypting Sensitive Environment Variables

Store all API keys, credentials, and tokens in environment files — **never in source code**.

Example .env file:

```
JWT_SECRET=supersecretkey
DB_PASSWORD=mydbpassword
SSL_KEY_PATH=/etc/ssl/private/server.key
```

Then load safely in your code:

```
import dotenv from "dotenv";
dotenv.config();
```

For production, use:

- **HashiCorp Vault**
- **AWS Secrets Manager**
- **Docker Secrets**
- or **Kubernetes Secret Manager**

6. Rate Limiting and Request Validation

To protect against denial-of-service attacks or brute force attempts, apply **rate limiting** on all endpoints.

Example using express-rate-limit:

```
import rateLimit from "express-rate-limit";

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 min
  max: 100, // limit each IP to 100 requests per window
});

app.use("/api", limiter);
```

Also, validate every incoming payload to prevent malformed requests:

```
if (!req.body || typeof req.body !== "object") {
  return res.status(400).json({ error: "Invalid input
format" });
}
```

7. Securing WebSocket or Persistent Connections

If your MCP server uses **WebSockets** for real-time interactions:

- Use **wss://** instead of **ws://**
- Require **token verification** before accepting connections
- Regularly **refresh or rotate tokens**

Example:

```
wss.on("connection", (ws, req) => {
  const token = req.headers.authorization?.split(" ")[1];
  if (!verifyToken(token)) {
    ws.close(4001, "Unauthorized");
    return;
  }
  console.log("🔒 Secure WebSocket connection
established");
});
```

8. Network-Level Security

Enforce additional security through infrastructure controls:

- Use **firewalls** to allow only necessary ports (443, 22, etc.).
- Use **VPNs or private networks** for internal components.
- Apply **IP whitelisting** for admin or monitoring endpoints.
- Monitor logs continuously with a tool like **Fail2Ban** or **CloudWatch**.

9. Certificate Rotation and Renewal

Certificates expire — and when they do, your clients lose access. Automate renewal using Let's Encrypt's **Certbot**:

```
sudo certbot renew --quiet
```

Schedule it with a cron job for continuous uptime:

```
0 0 * * * certbot renew --quiet
```

10. Testing Secure Connections

Before deploying, verify the integrity of your secure connection:

```
curl -v https://your-mcp-server.com
```

You should see:

```
* SSL connection using TLSv1.3
* Server certificate verified successfully
```

Also test from your MCP client:

```
openssl s_client -connect your-mcp-server.com:443
```

Ensure the certificate chain and CA signatures are correct.

11. Example Secure MCP Configuration

Example `.env` setup for a production-ready server:

```
PORT=443
NODE_ENV=production
JWT_SECRET=supersecurekey
SSL_CERT_PATH=/etc/ssl/certs/server.crt
SSL_KEY_PATH=/etc/ssl/private/server.key
LOG_LEVEL=info
RATE_LIMIT=100
```

Server initialization:

```
import fs from "fs";
import https from "https";
import app from "./app.js";

const options = {
  key: fs.readFileSync(process.env.SSL_KEY_PATH),
  cert: fs.readFileSync(process.env.SSL_CERT_PATH),
};

https.createServer(options, app).listen(process.env.PORT,
() => {
  console.log(`🔐 Secure MCP server running on port
${process.env.PORT}`);
});
```

Advanced Features and Customization

Using Custom Middleware

Middleware is one of the most powerful patterns in any modern server architecture — and **MCP servers are no exception**.

It acts as a **processing layer** between the incoming request and the final response, allowing developers to inject logic such as validation, logging, authentication, caching, or analytics without changing the core business code.
In short, middleware helps make your MCP server **modular, extensible, and secure**.

1. What Is Middleware?

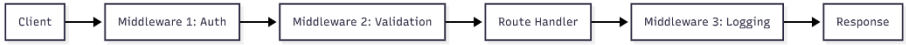
Middleware is a **function that runs before or after** your route handlers.
It receives the request, processes it, and either:

- Passes control to the next middleware (using `next()`), or
- Ends the response (by returning or sending a reply).

In the context of an MCP server (especially one built with Node.js and Express), middleware can:

- Inspect and modify requests/responses
- Enforce access control
- Handle errors or retries
- Transform data before sending it to the model or external APIs

Middleware flow example:



2. Why Use Middleware in MCP Servers

In MCP systems, middleware helps manage tasks like:

Middleware Type	Purpose
Authentication	Validate API keys, tokens, or client identity
Request Validation	Ensure incoming data follows the schema
Logging	Record each request and its duration
Rate Limiting	Prevent abuse or too many concurrent requests
Caching	Speed up repeated API calls
Response Shaping	Format the output for LLMs or clients

Error Handling	Capture and structure all errors consistently

This modular approach keeps your **tool routes** and **resource logic** clean and reusable.

3. Basic Middleware Example

A simple example that logs every request to your MCP server:

```
// middleware/logger.js
export function logger(req, res, next) {
  console.log(`[${new Date().toISOString()}] ${req.method}
${req.url}`);
  next(); // continue to next middleware or route
}
```

Then in your main server file:

```
import express from "express";
import { logger } from "./middleware/logger.js";

const app = express();
app.use(logger);

app.get("/getTime", (req, res) => {
  res.json({ time: new Date().toISOString() });
});

app.listen(3000, () => console.log("MCP Server running on
port 3000"));
```

Now every request is logged with a timestamp before reaching the route.

4. Middleware with Request Validation

Let's add a middleware that checks if the incoming request contains the proper structure before passing it to the tool.

```
// middleware/validateRequest.js
export function validateRequest(req, res, next) {
  if (!req.body || typeof req.body !== "object") {
    return res.status(400).json({ error: "Invalid request
body" });
  }

  if (!req.body.tool || !req.body.params) {
```

```
    return res.status(400).json({ error: "Missing tool or
params" });
  }

  next();
}
```

Then attach it to your routes:

```
import { validateRequest } from
"./middleware/validateRequest.js";
app.post("/runTool", validateRequest, runToolHandler);
```

This ensures only well-structured payloads reach your MCP logic.

5. Using Middleware for Authentication

You can easily secure specific endpoints with middleware.

```
// middleware/auth.js
export function authenticate(req, res, next) {
  const token = req.headers.authorization?.split(" ")[1];
  if (token !== process.env.API_TOKEN) {
    return res.status(403).json({ error: "Forbidden:
Invalid token" });
  }
  next();
}
```

Apply it selectively:

```
app.post("/secure/tool", authenticate, handleSecureTool);
```

This approach keeps your **security logic separated** from core tool implementation — making it easier to maintain and audit.

6. Middleware for Response Transformation

Sometimes, you'll want to format every response consistently — especially if your MCP server interacts with LLMs or structured agents.

```
// middleware/responseFormatter.js
export function responseFormatter(req, res, next) {
  const oldJson = res.json;
  res.json = function (data) {
    const formatted = {
```

```
      timestamp: new Date().toISOString(),
      status: "success",
      data,
    };
    oldJson.call(this, formatted);
  };
  next();
}
```

Then apply it globally:

```
app.use(responseFormatter);
```

Now, every response will include metadata and a standard format for easier parsing.

7. Middleware Execution Order

Order matters.

Middleware runs **in the sequence you define it** — so always structure it carefully.

Example of recommended order:

1. **Security** middleware (auth, rate limit)
2. **Validation** middleware
3. **Core route handlers**
4. **Logging** or **response formatting** middleware

Incorrect ordering can cause performance issues or bypass validation entirely.

8. Error-Handling Middleware

A special kind of middleware is designed specifically for catching errors.

```
// middleware/errorHandler.js
export function errorHandler(err, req, res, next) {
  console.error("Error:", err.stack);
  res.status(500).json({
    status: "error",
    message: err.message || "Internal server error",
  });
}
```

To use it:

```
app.use(errorHandler);
```

This ensures all runtime issues return structured, readable responses.

9. Combining Multiple Middlewares

You can chain multiple middlewares per route:

```
app.post(
  "/process",
  authenticate,
  validateRequest,
  responseFormatter,
  runProcessHandler
);
```

Execution flow:

Each layer can modify the request or stop it if something fails.

10. Custom Middleware in MCP Context

In more advanced MCP setups, middleware can be extended to:

- **Inject metadata** into the context before model calls
- **Track usage statistics** for tools or prompts
- **Wrap tool execution** to measure latency and success rates
- **Modify the model's system prompt** dynamically for debugging

This allows developers to customize **LLM interactions** without changing core tool logic.

11. Example: Middleware for Timing Tool Calls

```
// middleware/timer.js
export function timer(req, res, next) {
  const start = Date.now();
  res.on("finish", () => {
    const duration = Date.now() - start;
    console.log(`Tool executed in ${duration}ms`);
  });
  next();
}
```

Now each MCP tool route automatically reports execution time.

Best Practices

- Keep middleware **small and focused** — one purpose per file.
- Always call `next()` unless sending a response.
- Use **async/await** and try/catch inside middleware to handle async logic.
- Reuse middleware across routes where possible.
- For heavy middleware, use **caching or memoization** to avoid slowing down requests.

Error Handling Strategies

Building robust MCP servers requires anticipating and gracefully handling errors—both expected and unexpected. Error handling is not just about avoiding crashes; it's about maintaining clarity, consistency, and developer confidence.

1. Categorizing Errors

Errors in an MCP server typically fall into three types:

- **Client Errors (4xx):** Caused by invalid input, missing parameters, or unauthorized requests.
- **Server Errors (5xx):** Triggered by bugs, exceptions, or resource unavailability.
- **Network/Transport Errors:** Related to timeouts, disconnections, or JSON-RPC protocol mismatches.

Define clear error codes and consistent message structures for each.

2. Standard Error Response Format

Use structured JSON for predictable and debuggable errors:

```
{
  "error": {
    "code": 400,
    "message": "Invalid parameter: 'resourceId' is
required",
    "details": {
      "timestamp": "2025-10-15T12:34:56Z",
      "requestId": "abc123"
    }
  }
}
```

TIP: Always include contextual details like `requestId` or timestamps for easier traceability.

3. Global Error Middleware

Create a central middleware that catches and formats all errors before sending them back to the client:

```
server.use(async (req, res, next) => {
  try {
    await next();
  } catch (err) {
    console.error("Error caught:", err);
    res.send({
      error: {
        code: err.code || 500,
        message: err.message || "Internal Server Error",
      },
    });
  }
});
```

This ensures consistency across all routes, regardless of where the error originates.

4. Graceful Degradation

When possible, return PARTIAL OR FALLBACK RESULTS instead of failing completely.

Example:

```
try {
  const data = await fetchData();
} catch (e) {
  console.warn("Primary data fetch failed. Using cached
data.");
  const data = cache.get("backup");
}
```

5. Logging and Monitoring

Integrate structured logging tools such as **Winston**, **Pino**, or **Bunyan**:

- Log full stack traces in development.
- Log only summarized errors (with IDs) in production.
- Consider integrating **Sentry**, **Datadog**, or **OpenTelemetry** for observability.

6. Validating Before Executing

Always validate inputs at the earliest stage to prevent cascading errors:

```
if (!request.params.userId) {
  throw new Error("Missing userId parameter");
}
```

Combine this with schema validation using libraries like **Joi** or **Zod**.

7. Fail-Safe Defaults

When encountering unknown or unhandled conditions:

- Default to SAFE actions (e.g., deny access rather than grant it).
- Avoid exposing sensitive data in error messages.
- Ensure the system remains stable even after repeated failures.

Pro Tip: Treat every error as an opportunity for clarity. A well-structured error response not only helps debugging—it teaches clients how to interact with your server correctly.

Building Reusable Utilities

Reusable utilities are the **backbone of scalable MCP server design**. Instead of rewriting logic for logging, validation, or configuration, you can encapsulate these into well-structured utility modules that make your server cleaner, faster to maintain, and easier to extend.

1. Why Utilities Matter

Utilities help you:

- **Avoid repetition** (DRY principle — DON'T REPEAT YOURSELF)
- **Centralize common logic** (e.g., logging, error formatting)
- **Improve testability** (each utility can be unit tested independently)
- **Simplify onboarding** (developers understand consistent patterns)

2. Typical Utility Categories

Category	Purpose	Example Utility
Logging	Unified logs for requests & errors	`logger.js`
Validation	Schema-based input validation	`validateInput.js`
Error Handling	Consistent error formatting	`handleError.js`
Configuration	Load and manage environment vars	`config.js`
HTTP / Fetch	Simplified API calls	`fetchWrapper.js`
Security	Token verification or encryption	`authUtils.js`

3. Example: Logger Utility

A reusable logging utility makes it easy to track what's happening across modules.

```
// utils/logger.js
export const logger = {
  info: (msg, meta = {}) => console.log(`[INFO] ${msg}`,
meta),
  warn: (msg, meta = {}) => console.warn(`[WARN] ${msg}`,
meta),
  error: (msg, meta = {}) => console.error(`[ERROR]
${msg}`, meta),
};
```

Usage:

```
import { logger } from "./utils/logger.js";

logger.info("Server started");
logger.error("Database connection failed", { retrying: true
});
```

4. Example: Validation Utility

Keep input validation separate so routes stay clean and readable.

```
// utils/validateInput.js
export function validateInput(schema, data) {
  const result = schema.safeParse(data);
  if (!result.success) {
    throw new Error(`Validation failed:
${result.error.message}`);
  }
  return result.data;
}
```

Usage:

```
import { z } from "zod";
import { validateInput } from "./utils/validateInput.js";

const requestSchema = z.object({
  userId: z.string().uuid(),
  limit: z.number().min(1).max(100),
});

const input = validateInput(requestSchema, req.params);
```

5. Example: Auth Utility

Centralize token validation for secure access control.

```
// utils/auth.js
export function verifyToken(token) {
  if (!token || token.split(".").length !== 3) {
    throw new Error("Invalid token format");
  }
  // decode or verify with secret key here
  return true;
}
```

6. Configuration Utility

Use a single module to load environment variables with defaults.

```
// utils/config.js
import dotenv from "dotenv";
dotenv.config();

export const config = {
  PORT: process.env.PORT || 3000,
  API_KEY: process.env.API_KEY,
  ENV: process.env.NODE_ENV || "development",
};
```

7. Organizing Utilities

A clean project structure might look like this:

```
src/
├─ server/
│  ├─ index.js
│  ├─ routes/
│  ├─ middlewares/
├─ utils/
│  ├─ logger.js
│  ├─ validateInput.js
│  ├─ config.js
│  ├─ auth.js
│  └─ handleError.js
```

This keeps the core logic focused while utilities remain easily accessible and reusable.

8. Testing Utilities Independently

Because utilities are standalone modules, you can write fast, focused tests for each.

Example using Jest:

```
import { verifyToken } from "../utils/auth.js";

test("rejects invalid token format", () => {
  expect(() =>
verifyToken("invalidToken")).toThrow("Invalid token
format");
});
```

Adding Rate Limiting and Request Queues

As your MCP server grows and begins handling more clients or requests per second, performance and reliability become crucial. Two mechanisms—**rate limiting** and **request queuing**—help ensure your server remains stable, fair, and secure even under heavy load.

1. What Is Rate Limiting?

Rate limiting controls how many requests a client can make within a specific time window.

It prevents abuse (e.g., spamming API calls) and protects system resources.

Example use cases:

- Avoiding denial-of-service (DoS) attacks.
- Protecting expensive or external API calls.
- Enforcing fair usage between clients.

2. Common Rate-Limiting Strategies

Strategy	Description	Example Use Case
Fixed Window	Allows N requests per time window (e.g., 100 requests per minute).	Simple APIs
Sliding Window	Uses a moving window to track usage more precisely.	Real-time systems
Token Bucket	Tokens are refilled over time; each request consumes one.	Streaming or high-frequency requests

Leaky Bucket	Requests enter a queue and are processed at a steady rate.	Stable throughput under bursts

3. Implementing Simple Rate Limiting (Example in Node.js)

Here's a lightweight implementation using a **token bucket pattern**:

```
// utils/rateLimiter.js
const rateLimiters = new Map();

export function rateLimiter(clientId, limit = 5, interval =
10000) {
  const now = Date.now();
  if (!rateLimiters.has(clientId)) {
    rateLimiters.set(clientId, []);
  }

  const timestamps = rateLimiters.get(clientId).filter(t =>
now - t < interval);

  if (timestamps.length >= limit) {
    throw new Error("Rate limit exceeded. Please try again
later.");
  }

  timestamps.push(now);
  rateLimiters.set(clientId, timestamps);
}
```

Usage:

```
import { rateLimiter } from "./utils/rateLimiter.js";

try {
  rateLimiter("user-123", 10, 60000); // 10 requests/minute
  console.log("Request allowed");
} catch (err) {
  console.error(err.message);
}
```

4. Using a Third-Party Library

For production-grade applications, use established libraries such as:

- **Express-rate-limit** (for Express.js servers)

- **Bottleneck** (for general task throttling)
- **Redis-based rate limiting** (for distributed systems)

Example using bottleneck:

```
import Bottleneck from "bottleneck";

const limiter = new Bottleneck({
  maxConcurrent: 2,
  minTime: 200,
});

async function fetchData() {
  console.log("Fetching data...");
}

limiter.schedule(fetchData);
```

This ensures only 2 concurrent tasks run, spacing each by at least 200ms.

5. What Are Request Queues?

Request queues temporarily hold incoming requests when your system is at capacity. They allow for **graceful degradation** rather than outright rejection.

Benefits:

- Smooths traffic spikes.
- Prevents overload on databases or external APIs.
- Maintains consistent response times.

6. Simple Request Queue Example

```
// utils/requestQueue.js
class RequestQueue {
  constructor(limit = 5) {
    this.queue = [];
    this.active = 0;
    this.limit = limit;
  }

  async enqueue(task) {
    return new Promise((resolve, reject) => {
      this.queue.push({ task, resolve, reject });
      this.process();
```