Orders Circuit Breaker Demo (Node.js + Express)

Resilience demo: Circuit Breaker pattern to stop cascading failures and provide fast fallback

Index

1. Project Overview

This application demonstrates the Circuit Breaker pattern: when a downstream dependency becomes slow or fails repeatedly, the server temporarily stops calling it and returns a fast failure or fallback response. This prevents cascading failures and keeps the Orders API responsive.

Scenario

Imagine the Orders API calls a payment service. If payment starts timing out, every request waits and threads/connections pile up. A circuit breaker 'opens' after repeated failures so requests fail fast (or use a fallback) until the dependency recovers.

2. Learning Outcomes

- Explain circuit breaker states: CLOSED → OPEN → HALF-OPEN → CLOSED.
- Configure thresholds: failure percentage / consecutive failures, timeouts, and reset window.
- Implement fallbacks (optional) to degrade gracefully.
- Debug and verify circuit breaker behavior under failures and recovery.

3. Tech Stack

package.json:

```
{
  "name": "orders-circuit-breaker-demo",
  "version": "1.0.0",
  "type": "commonjs",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.19.2",
    "mongoose": "^8.5.2"
  }
}
```

4. Folder Structure

```
- circuitBreaker.js
- package.json
- server.js
  - Order.js
  - orders.js
```

5. High-Level Architecture

Client → Express routes → Circuit Breaker wrapper → downstream call → response

When the breaker is OPEN, the downstream call is skipped and a fast response is returned.

6. Circuit Breaker Pattern in This App

Circuit breaker is like an electrical fuse. When the circuit overloads (too many failures/timeouts), the fuse opens to protect the system. After a cool-down window, it allows a few test requests (HALF-OPEN) to check if the dependency recovered.

State transitions (mental model):

- CLOSED: All calls go through; failures are tracked.
- OPEN: Calls are blocked; fail fast or fallback.
- HALF-OPEN: Allow limited trial calls; if they succeed → CLOSED; if fail → OPEN.

Common configuration knobs:

- timeout: maximum time allowed per downstream call
- failure threshold: how many failures or what error percentage trips OPEN
- reset timeout: how long breaker stays OPEN before trying HALF-OPEN
- fallback: response used when OPEN (optional)

Circuit breaker implementation files detected:

- circuitBreaker.js

Circuit breaker-related snippets found in your code:

circuitBreaker.js

```
// Simple Circuit Breaker implementation
class CircuitBreaker {
  constructor({ failureThreshold = 3, resetTimeout = 10000 }) {
    this.failureThreshold = failureThreshold;
```

package.json

```
{
  "name": "orders-circuit-breaker-demo",
  "version": "1.0.0",
  "type": "commonjs",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.19.2",
    "mongoose": "^8.5.2"
  }
}
```

server.js

```
app.use("/orders", orderRoutes);

app.listen(3000, () => {
  console.log("Orders Circuit Breaker Demo running on
http://localhost:3000");
});
```

routes/orders.js

```
const router = express.Router();

// Circuit breaker protecting payment service
const paymentBreaker = new CircuitBreaker({
  failureThreshold: 3,
  resetTimeout: 10000
```

7. API Endpoints

Base URL: http://localhost:3000 (or as configured)

Method	Path	Source File	Notes
POST	/orders/	routes/orders.js	
GET	/orders/:id	routes/orders.js	

8. How to Run + Quick Tests

Install and run:

```
npm install
npm run dev
```

Smoke test:

Create multiple orders quickly (open 3 terminals and run simultaneously)

```
curl -X POST http://localhost:3000/orders \
 -H "Content-Type: application/json" \
 -d '{"product":"Laptop","amount":50000}'
```

Trip the circuit (test idea):

```
# Cause downstream failures (e.g., stop the dependency or point to a bad URL)
# Then call the protected endpoint multiple times and observe:
# - initial slow/failing responses (CLOSED)
# - then fast failures/fallback (OPEN)
# After resetTimeout, try again to see HALF-OPEN behavior.
```

9. Common Mistakes

Treating circuit breaker like retry

Circuit breaker stops calls; retry repeats calls. Retrying during outage can make it worse.

No timeouts

Without timeouts, requests hang and the breaker does not learn fast enough.

Threshold too aggressive

Trips too often, causing unnecessary OPEN state under minor blips.

No visibility/metrics

Without state change logs/metrics, you can’t confirm OPEN/HALF-OPEN behavior.

Bad fallback design

Fallback should be safe and minimal; not another slow call that can also fail.

10. Debugging Techniques

- Log circuit breaker state transitions (CLOSED/OPEN/HALF-OPEN) with timestamps.
- Log timeout vs non-timeout failures separately (timeouts usually indicate latency issues).
- During OPEN, confirm the downstream call is not executed (no outbound request logs).
- Verify resetTimeout: wait the cool-down period and test if HALF-OPEN allows trial calls.
- Use controlled failure injection (stop dependency, add artificial delay) to reproduce trips.

Appendix A: Full Source Code

circuitBreaker.js

```
// Simple Circuit Breaker implementation
class CircuitBreaker {
  constructor({ failureThreshold = 3, resetTimeout = 10000 }) {
    this.failureThreshold = failureThreshold;
    this.resetTimeout = resetTimeout;

    this.failureCount = 0;
    this.state = "CLOSED"; // CLOSED | OPEN | HALF_OPEN
    this.nextAttempt = Date.now();
  }

  async execute(action) {
    if (this.state === "OPEN") {
      if (Date.now() > this.nextAttempt) {
        this.state = "HALF_OPEN";
      } else {
        throw new Error("Circuit breaker is OPEN");
      }
    }

    try {
      const result = await action();
      this.success();
      return result;
    } catch (err) {
      this.failure();
      throw err;
    }
  }

  success() {
    this.failureCount = 0;
    this.state = "CLOSED";
  }

  failure() {
    this.failureCount += 1;

    if (this.failureCount >= this.failureThreshold) {
      this.state = "OPEN";
      this.nextAttempt = Date.now() + this.resetTimeout;
      console.log("[CircuitBreaker] OPEN");
    }
  }
}

module.exports = CircuitBreaker;
```

package.json

```
{
  "name": "orders-circuit-breaker-demo",
  "version": "1.0.0",
  "type": "commonjs",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.19.2",
    "mongoose": "^8.5.2"
  }
}
```

server.js

```

const express = require("express");
const mongoose = require("mongoose");
const orderRoutes = require("./routes/orders");

const app = express();
app.use(express.json());
```

```
mongoose.connect("mongodb://localhost:27017/orders_circuit_breaker_demo")
  .then(() => console.log("MongoDB connected"))
  .catch(err => console.error(err));

app.use("/orders", orderRoutes);

app.listen(3000, () => {
  console.log("Orders Circuit Breaker Demo running on
http://localhost:3000");
});
```

models/Order.js

```
const mongoose = require("mongoose");

const OrderSchema = new mongoose.Schema(
  {
    product: String,
    amount: Number,
    status: String
  },
  { timestamps: true }
);

module.exports = mongoose.model("Order", OrderSchema);
```

routes/orders.js

```
const express = require("express");
const Order = require("../models/Order");
const CircuitBreaker = require("../circuitBreaker");

const router = express.Router();

// Circuit breaker protecting payment service
const paymentBreaker = new CircuitBreaker({
  failureThreshold: 3,
  resetTimeout: 10000
});

// Simulated external payment service
async function fakePaymentService(orderId) {
  console.log("Calling payment service for", orderId);

  // Simulate failure 70% of the time
  if (Math.random() < 0.7) {
    throw new Error("Payment service failed");
  }

  return "PAID";
}
```

```
// CREATE ORDER (Circuit Breaker protected)
router.post("/", async (req, res) => {
  try {
    const order = await Order.create({
      product: req.body.product,
      amount: req.body.amount,
      status: "CREATED"
    });

    await paymentBreaker.execute(async () => {
      const status = await fakePaymentService(order._id);
      order.status = status;
      await order.save();
    });

    res.status(201).json(order);
  } catch (err) {
    res.status(503).json({
      error: err.message,
      breakerState: paymentBreaker.state
    });
  }
});

// READ ORDER
router.get("/:id", async (req, res) => {
  const order = await Order.findById(req.params.id);
  if (!order) return res.status(404).json({ error: "Order not found" });
  res.json(order);
});

module.exports = router;
```