**Circuit Breaker**

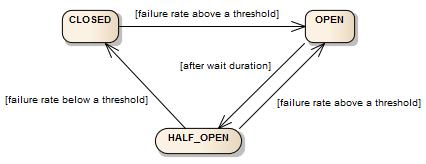
**Intro:**

**Circuit Breaker** is a design pattern used to protect your service from calling a **failing or slow system** repeatedly.

**Why use?**

1. Prevent cascading failures
2. Improve stability
3. Help services recover
4. Avoid wasting resources

**This is the Circuit Breaker diagram.**



***Diagram: Circuit Breaker***

There is total six states in this.

1. CLOSED STATE
2. OPEN STATE
3. HALF\_OPEN
4. FORCED\_OPEN STATE
5. DISABLED STATE
6. METRICS\_ONLY STATE

In the given image, only three states are shown: CLOSED, OPEN, and HALF-OPEN. Apart from these, FORCED-OPEN, DISABLED, and METRICS-ONLY states also exist, but they internally work with or transition around these main states (CLOSED, OPEN, HALF-OPEN)

Refer this example:

Request

Service A

Service B

Response

**For example, in the given diagram there are two services: Service A and Service B.**  
Suppose Service A is making requests to Service B.  
Now, if Service B goes down (is not responding), Service A will continue sending requests to Service B.

In this case, several problems can occur, such as:

* **Resource wastage** (CPU, memory, threads on Service A are stuck waiting)
* **Increased load on Service B** (if it is partially recovering, new requests make it worse)
* **Thread pool exhaustion** on Service A (causing Service A itself to slow down or crash)
* **Increased latency** for Service A’s users
* **Cascading failure** — one failing service brings down other dependent services

Let’s discuss all the states one by one.

**CLOSED STATE:**

1. In CLOSED state, everything is normal.
2. Service A is calling Service B — all requests are going through to Service B.
3. Circuit Breaker is just **monitoring** if Service B is healthy.

* If Service B is responding successfully → Circuit stays **CLOSED** (normal)
* If Service B starts failing too much → Circuit will move to **OPEN**.

**Example:**  
Service A calls Service B 100 times (with this CircuitBreaker config):

* + failureRateThreshold: 50
  + slidingWindowSize: 10

1. Out of 100 calls, 60 failed -> Failure rate = **60%** (> 50% threshold)
2. Circuit Breaker will now move to **OPEN state** —
3. It will block further calls from Service A to Service B for some time.
4. In **CLOSED**, nothing is blocked — Service A continues calling Service B.

**OPEN STATE (with Service A and B):**

1. If Service B starts failing too much (for example: > 50% failures),  
   the Circuit Breaker in Service A will move to **OPEN state**.

* **Service A will stop sending calls to Service B**.
* Any new call from Service A will be **blocked immediately**.
* Service A can either:
  + return a fallback response
  + or throw an exception (depending on config)

**HALF-OPEN STATE (with Service A and B):**

* **After the Circuit Breaker in Service A stays in OPEN state for a certain time (like 5 seconds),  
  it moves to the HALF-OPEN state.**
* **Only a few test calls** from Service A are allowed to go to Service B.  
  (example: 2 test calls)
* The goal is to **check if Service B has recovered**.

**What happens in HALF-OPEN?**

| If test calls succeed (no failures) → | Circuit Breaker moves to CLOSED again (normal) |  
| If test calls fail → | Circuit Breaker goes back to OPEN |

**After 5 seconds in OPEN state:**

1. Service A allows 2 real calls to Service B
2. If both succeed → Circuit goes to CLOSED (recovery successful)
3. If even 1 fails → Circuit goes back to OPEN

| **State** | **What happens** |
| --- | --- |
| HALF-OPEN | Service A allows limited test calls to Service B to check if it's healthy again |

**FORCED-OPEN STATE (with Service A and B):**

FORCED-OPEN is a manual state — set by developer or admin.

You can force the CircuitBreaker in Service A to stay **OPEN**, no matter what the failure rate is.

**What happens in FORCED-OPEN?**

* **All calls from Service A to Service B are blocked**.
* No test calls allowed (unlike HALF-OPEN).
* The circuit will stay FORCED-OPEN until **you manually change it**.

**Why use FORCED-OPEN?**

If you know Service B is having a big problem or under maintenance —  
You can **force Circuit Breaker to open**, so Service A stops sending requests.

| **State** | **What happens** |
| --- | --- |

|  |  |
| --- | --- |
| FORCED-OPEN | 🡪 All calls blocked manually — until admin allows it to recover |

**DISABLED STATE (with Service A and B):**

* **DISABLED** means the CircuitBreaker is completely **turned off**.
  + **All calls from Service A go to Service B** — no blocking, no fallback.
  + Circuit Breaker **does not open or close** — it is inactive.
  + **Metrics are still recorded** — so you can observe, but no protection is applied.

**Why use DISABLED?**

* If you want to temporarily **disable Circuit Breaker** (for example, during testing)
* If Service B is stable and you want to **monitor only**
* If fallback logic is under development, but you want to collect stats

| **State** | **What happens** |
| --- | --- |
| DISABLED | 🡪 All calls go through Circuit Breaker never blocks — metrics still collected |

**METRICS\_ONLY STATE (with Service A and B):**

* In METRICS\_ONLY, the Circuit Breaker behaves like DISABLED, but with a purpose:
* **All calls from Service A go to Service B** — no blocking.
* Circuit Breaker **only collects metrics** —
  + number of successes
  + number of failures
  + current failure rate
  + state transitions (OPEN/CLOSED would have happened, but they won’t block calls)

**Difference vs DISABLED:**

| **State** | **What happens** |
| --- | --- |
| **DISABLED** | No logic at all — no sliding window — only raw counters (calls count). |
| **METRICS\_ONLY** | Full sliding window logic runs — failure % is calculated — **but no calls are blocked** — you can "see" what would happen if it was active. |

**Why use METRICS\_ONLY?**

✅ When adding CircuitBreaker for the first time:  
→ You want to **observe the failure rate**  
→ But you don’t want to accidentally block traffic.

✅ You can "test" the config:  
→ If failureRateThreshold = 50% — how often would the CircuitBreaker have opened?  
→ You can tune settings safely.

**Config property**

| **Property Name** | **Description** | **Example Value / Notes** |
| --- | --- | --- |
| slidingWindowType | Type of window: COUNT or TIME based | COUNT\_BASED or TIME\_BASED |
| slidingWindowSize | Size of the window (number of calls or seconds) | 10 |
| minimumNumberOfCalls | Minimum calls before failure % is calculated | 5 |
| failureRateThreshold | Failure % to trigger OPEN state | 50 (%) |
| waitDurationInOpenState | How long to stay OPEN | 5s |
| permittedNumberOfCallsInHalfOpenState | Number of test calls in HALF-OPEN | 2 |
| automaticTransitionFromOpenToHalfOpenEnabled | Auto move from OPEN to HALF-OPEN | true or false |
| slowCallRateThreshold | % of slow calls considered failure | 50 (%) |
| slowCallDurationThreshold | Duration above which call is slow | 2s |
| recordExceptions | Exceptions to treat as failure | List of Exception classes |
| ignoreExceptions | Exceptions to **ignore** as failure | List of Exception classes |
| maxWaitDurationInHalfOpenState | Optional: Max time to wait in HALF-OPEN | 10s |
| writableStackTraceEnabled | Enable/disable stack trace generation | true or false |
| registerHealthIndicator | Add CB status to /actuator/health | true or false |
| recordFailurePredicate | Advanced: function to define failure logic | Java Predicate Bean |