Graceful Shutdown of Servers

1. The Real Problem This Exists to Solve

When you restart or deploy a server, in-flight requests can be killed mid-execution, causing failed requests, data corruption, and poor user experience. Graceful shutdown ensures all active requests complete before the process exits, preventing data loss and maintaining service reliability during deployments.

Real production scenario:

- E-commerce API handling 1,000 requests/second
- Deploy new version (rolling restart)
- **Without graceful shutdown:**
  - SIGTERM signal sent to process
  - Process exits immediately
  - 50 active requests killed mid-execution
  - Payment partially processed (charged but order not created)
  - Database transaction left open
  - User sees: 500 Internal Server Error
  - Support tickets flood in
  - Financial reconciliation nightmare
- **With graceful shutdown:**
  - SIGTERM signal sent to process
  - Server stops accepting new requests
  - Load balancer redirects new traffic to other instances
  - Server waits for 50 active requests to complete
  - All requests finish successfully
  - Database connections closed cleanly
  - Process exits with code 0
  - Zero failed requests
  - Users unaware of deployment

The fundamental problem: Servers handle stateful operations (database transactions, file uploads, API calls). Abruptly killing processes mid-execution leaves systems in inconsistent states. Graceful shutdown provides time for cleanup, ensuring zero-downtime deployments and preventing data corruption.

Without graceful shutdown:

- Failed requests during deployments
- Data corruption (partial writes)
- Orphaned database connections
- User-visible errors
- Lost revenue

With graceful shutdown:

- Zero failed requests
- Clean database transactions
- No orphaned connections
- Seamless deployments
- Happy users

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: No Shutdown Handling (Immediate Exit)

```
// Incorrect: Server exits immediately on SIGTERM
import express from 'express';

const app = express();

app.post('/api/orders', async (req, res) => {
  // Start processing order
  await createOrder(req.body);

  // Charge credit card
  await chargeCard(req.body.payment);

  // Send confirmation email
  await sendEmail(req.body.email);

  res.json({ success: true });
});

const server = app.listen(3000);

// No SIGTERM handler!
// When SIGTERM arrives, process exits immediately
```

Why it seems reasonable:

- Simple, no extra code
- Operating system handles cleanup
- Works fine in development

How it breaks:

```
Timeline:
T0: Request arrives: POST /api/orders
T1: createOrder() starts (writes to database)
T2: SIGTERM signal received (deployment triggered)
T3: Process exits immediately (default behavior)
T4: createOrder() killed mid-execution
T5: Database transaction left uncommitted
T6: chargeCard() never called
T7: User charged? No. Order created? Partial. State: BROKEN

Result:
- Order in database (half-created, invalid state)
- Payment not processed
- Email not sent
- User sees: Connection reset error
```

Production symptoms:

- Spike in 502/503 errors during deployments
- Orphaned database connections (max connections exhausted)
- Partial orders in database
- Support: "My order failed, but I see it in the system?"

❌ Incorrect Approach #2: Fixed Delay (Arbitrary Wait)

```
// Incorrect: Wait fixed time, then exit
process.on('SIGTERM', () => {
  console.log('SIGTERM received, waiting 5 seconds...');

  setTimeout(() => {
    console.log('Exiting now');
    process.exit(0);
  }, 5000);
});

const server = app.listen(3000);
```

Why it seems reasonable:

- Gives requests time to finish
- Simple to implement
- Better than immediate exit

How it breaks:

```
Scenario 1: Long-running request
T0: Request starts (file upload, takes 30 seconds)
T1: SIGTERM received
T2: Wait 5 seconds
T6: Exit (request still running!)
T7: Upload killed at 17% complete

Scenario 2: No active requests
T0: SIGTERM received
T1: No active requests (idle server)
T2: Wait 5 seconds (wasted time)
T6: Exit

Problems:
- 5 seconds not enough for long requests
- 5 seconds wasted if no requests
- Still kills active requests
```

Production symptoms:

```
Monitoring:
- P99 request duration: 2s
- Shutdown timeout: 5s
```

```
- Failed requests: 0.5% (long-tail requests >5s killed)
- Deployment takes 5s per instance (wasted time on idle servers)
```

❌ Incorrect Approach #3: Stop Accepting But Don't Wait

```javascript
// Incorrect: Close server but don't wait for active requests
process.on('SIGTERM', () => {
  console.log('SIGTERM received, closing server...');

  server.close(() => {
    console.log('Server closed');
    process.exit(0);
  });

  // Exits immediately! (no waiting)
});
```

Why it seems reasonable:

- Calls server.close() (looks correct)
- Stops accepting new connections
- Cleaner than immediate exit

How it breaks:

```javascript
server.close(() => {
  // This callback fires when server STOPS ACCEPTING new connections
  // NOT when all active requests finish!
  process.exit(0); // Active requests still running!
});
```

Production symptoms:

```
T0: 10 active requests
T1: SIGTERM received
T2: server.close() called
T3: Server stops accepting new requests
T4: Callback fires immediately (server closed)
T5: process.exit(0) called
T6: 10 active requests KILLED

Result:
- All in-flight requests fail
- Same as no shutdown handling
```

❌ Incorrect Approach #4: Wait Forever (No Timeout)

```javascript
// Incorrect: Wait for requests, but no timeout
process.on('SIGTERM', async () => {
  console.log('SIGTERM received, waiting for requests...');
```

```
  server.close(() => {
    console.log('All requests finished');
    process.exit(0);
  });

  // No timeout! Will wait forever if requests hang
});
```

Why it seems reasonable:

- Waits for all requests to complete
- No arbitrary timeout
- Ensures clean shutdown

How it breaks:

```
Scenario: Stuck request
T0: Request arrives (buggy code, infinite loop)
T1: SIGTERM received
T2: Wait for request to finish
T3: Request never finishes (stuck)
T4: Server never shuts down
T5: Orchestrator (K8s) waits 30s (termination grace period)
T30: SIGKILL sent (force kill)
T31: Process killed forcefully

Result:
- Graceful shutdown failed
- Forced termination
- Partial data corruption
- Same as no shutdown handling
```

Production symptoms:

```
Kubernetes logs:
"Termination grace period exceeded, sending SIGKILL"

Monitoring:
- Pod termination time: 30s (should be 5-10s)
- Deployment slow (waiting for stuck pods)
```

❌ Incorrect Approach #5: Disconnect Database Immediately

```
// Incorrect: Close database before requests finish
process.on('SIGTERM', async () => {
  console.log('SIGTERM received');

  // Close database immediately
  await db.close();

  // Now wait for requests (but they need the database!)
```

```
  server.close(() => {
    process.exit(0);
  });
});
```

Why it seems reasonable:

- Cleans up resources
- Closes connections
- Orderly shutdown

How it breaks:

```
T0: 5 active requests (using database)
T1: SIGTERM received
T2: db.close() called
T3: Database connections closed
T4: Active requests try to query database
T5: Error: "Connection pool closed"
T6: 5 requests fail

Result:
- Requests fail during shutdown
- Users see errors
- Defeats purpose of graceful shutdown
```

3. Correct Mental Model (How It Actually Works)

Graceful shutdown is a multi-phase process: stop accepting new requests, wait for active requests to complete (with timeout), close resources, then exit. The goal is zero failed requests during deployment.

The Shutdown Sequence

```
1. Receive SIGTERM signal
2. Mark server as "shutting down"
3. Stop accepting new connections (server.close())
4. Wait for active requests to complete
5. If timeout exceeded, force shutdown
6. Close database connections
7. Close other resources (Redis, queues)
8. Exit process (code 0)
```

State Transitions

```
Running → Shutting Down → Draining → Closed → Exited

Running: Accept new requests, process existing
Shutting Down: Reject new requests, process existing
Draining: Wait for active requests to finish
Closed: All requests done, close resources
Exited: Process terminated
```

Health Check Coordination

```
Before shutdown:
  GET /health → 200 OK (load balancer sends traffic)

During shutdown:
  GET /health → 503 Service Unavailable (load balancer stops traffic)
  Existing requests: Still processing
  New requests: Rejected (503)
```

4. Correct Design & Algorithm

Strategy 1: Server.close() with Timeout

```
process.on('SIGTERM', () => {
  const timeout = 30000; // 30 seconds

  server.close(() => {
    // All connections closed
    cleanupResources();
    process.exit(0);
  });

  // Force exit if timeout exceeded
  setTimeout(() => {
    console.error('Forced shutdown (timeout)');
    process.exit(1);
  }, timeout);
});
```

Strategy 2: Track Active Requests

```
let activeRequests = 0;

app.use((req, res, next) => {
  activeRequests++;
  res.on('finish', () => activeRequests--);
  next();
});

process.on('SIGTERM', async () => {
  while (activeRequests > 0) {
    await sleep(100);
  }
  process.exit(0);
});
```

Strategy 3: Health Check + Drain

```
let isShuttingDown = false;

app.get('/health', (req, res) => {
  if (isShuttingDown) {
    res.status(503).json({ status: 'shutting down' });
  } else {
    res.status(200).json({ status: 'healthy' });
  }
});

process.on('SIGTERM', () => {
  isShuttingDown = true;
  // Load balancer removes this instance
  setTimeout(shutdown, 10000); // Drain for 10s
});
```

5. Full Production-Grade Implementation

```
import express from 'express';
import { Pool } from 'pg';
import http from 'http';

/**
 * Graceful shutdown manager
 */
class GracefulShutdown {
  private isShuttingDown = false;
  private activeRequests = 0;
  private shutdownTimeout: number;
  private connections = new Set<any>();

  constructor(
    private server: http.Server,
    private resources: ShutdownResource[],
    options: { timeout?: number } = {}
  ) {
    this.shutdownTimeout = options.timeout || 30000; // 30s default
    this.setupHandlers();
  }

  /**
   * Setup signal handlers
   */
  private setupHandlers() {
    process.on('SIGTERM', () => this.shutdown('SIGTERM'));
    process.on('SIGINT', () => this.shutdown('SIGINT'));

    // Handle uncaught errors
    process.on('uncaughtException', (error) => {
```

```
    console.error('Uncaught exception:', error);
    this.shutdown('uncaughtException');
  });

  process.on('unhandledRejection', (reason) => {
    console.error('Unhandled rejection:', reason);
    this.shutdown('unhandledRejection');
  });
}

/**
 * Track active requests
 */
trackRequest(req: express.Request, res: express.Response) {
  this.activeRequests++;

  res.on('finish', () => {
    this.activeRequests--;
  });

  res.on('close', () => {
    this.activeRequests--;
  });
}

/**
 * Check if shutting down
 */
isShutdown(): boolean {
  return this.isShuttingDown;
}

/**
 * Get active request count
 */
getActiveRequests(): number {
  return this.activeRequests;
}

/**
 * Initiate graceful shutdown
 */
private async shutdown(signal: string) {
  if (this.isShuttingDown) {
    return; // Already shutting down
  }

  console.log(`[Shutdown] Received ${signal}, starting graceful shutdown...`);
  this.isShuttingDown = true;

  // Start timeout timer
  const forceExitTimer = setTimeout(() => {
```

```
    console.error(
      `[Shutdown] Timeout exceeded (${this.shutdownTimeout}ms), forcing exit`
    );
    console.error(`[Shutdown] ${this.activeRequests} requests were still active`);
    process.exit(1);
  }, this.shutdownTimeout);

  try {
    // Phase 1: Stop accepting new connections
    console.log('[Shutdown] Phase 1: Stopping new connections...');
    await this.closeServer();

    // Phase 2: Wait for active requests to complete
    console.log(
      `[Shutdown] Phase 2: Waiting for ${this.activeRequests} active requests...`
    );
    await this.waitForRequests();

    // Phase 3: Close resources (DB, Redis, etc.)
    console.log('[Shutdown] Phase 3: Closing resources...');
    await this.closeResources();

    // Phase 4: Exit cleanly
    console.log('[Shutdown] Shutdown complete, exiting');
    clearTimeout(forceExitTimer);
    process.exit(0);
  } catch (error) {
    console.error('[Shutdown] Error during shutdown:', error);
    clearTimeout(forceExitTimer);
    process.exit(1);
  }
}

/**
 * Close HTTP server (stop accepting new connections)
 */
private closeServer(): Promise<void> {
  return new Promise((resolve) => {
    this.server.close(() => {
      console.log('[Shutdown] Server closed (no longer accepting connections)');
      resolve();
    });
  });
}

/**
 * Wait for active requests to complete
 */
private async waitForRequests(): Promise<void> {
  const startTime = Date.now();
  const checkInterval = 100; // Check every 100ms
```

```
    while (this.activeRequests > 0) {
      const elapsed = Date.now() - startTime;

      if (elapsed > this.shutdownTimeout - 5000) {
        console.warn(
          `[Shutdown] Still ${this.activeRequests} active requests after
${elapsed}ms`
        );
      }

      await new Promise((resolve) => setTimeout(resolve, checkInterval));
    }

    const elapsed = Date.now() - startTime;
    console.log(`[Shutdown] All requests completed in ${elapsed}ms`);
  }

  /**
   * Close external resources
   */
  private async closeResources(): Promise<void> {
    for (const resource of this.resources) {
      try {
        console.log(`[Shutdown] Closing ${resource.name}...`);
        await resource.close();
        console.log(`[Shutdown] ${resource.name} closed`);
      } catch (error) {
        console.error(`[Shutdown] Error closing ${resource.name}:`, error);
      }
    }
  }
}

/**
 * Resource interface for cleanup
 */
interface ShutdownResource {
  name: string;
  close: () => Promise<void>;
}

/**
 * Database resource
 */
class DatabaseResource implements ShutdownResource {
  name = 'Database';

  constructor(private pool: Pool) {}

  async close(): Promise<void> {
    await this.pool.end();
  }
```

```
}

/**
 * Redis resource
 */
class RedisResource implements ShutdownResource {
  name = 'Redis';

  constructor(private client: any) {}

  async close(): Promise<void> {
    await this.client.quit();
  }
}

// Express app setup
const app = express();
const server = http.createServer(app);

// Database connection
const db = new Pool({
  host: 'localhost',
  database: 'myapp',
  max: 20,
});

// Initialize graceful shutdown
const shutdown = new GracefulShutdown(
  server,
  [new DatabaseResource(db)],
  { timeout: 30000 }
);

/**
 * Middleware: Track active requests
 */
app.use((req, res, next) => {
  shutdown.trackRequest(req, res);
  next();
});

/**
 * Middleware: Reject requests during shutdown
 */
app.use((req, res, next) => {
  if (shutdown.isShutdown()) {
    res.status(503).json({
      error: 'Server is shutting down',
      retryAfter: 10,
    });
    return;
  }
```

```
  next();
});

/**
 * Health check endpoint
 */
app.get('/health', (req, res) => {
  if (shutdown.isShutdown()) {
    return res.status(503).json({
      status: 'shutting down',
      activeRequests: shutdown.getActiveRequests(),
    });
  }

  res.status(200).json({
    status: 'healthy',
    activeRequests: shutdown.getActiveRequests(),
  });
});

/**
 * Liveness probe (always healthy unless crashed)
 */
app.get('/healthz/liveness', (req, res) => {
  res.status(200).json({ status: 'alive' });
});

/**
 * Readiness probe (unhealthy during shutdown)
 */
app.get('/healthz/readiness', (req, res) => {
  if (shutdown.isShutdown()) {
    return res.status(503).json({ status: 'not ready' });
  }

  res.status(200).json({ status: 'ready' });
});

/**
 * Example: Long-running endpoint
 */
app.post('/api/orders', async (req, res) => {
  try {
    // Simulate long-running operation
    console.log('[Order] Creating order...');
    await createOrder(req.body);

    console.log('[Order] Charging payment...');
    await chargePayment(req.body.payment);

    console.log('[Order] Sending confirmation...');
    await sendConfirmation(req.body.email);
```

```
    res.json({ success: true });
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

async function createOrder(data: any): Promise<void> {
  await db.query('INSERT INTO orders (data) VALUES ($1)', [JSON.stringify(data)]);
  await new Promise((resolve) => setTimeout(resolve, 2000)); // 2s
}

async function chargePayment(payment: any): Promise<void> {
  // Simulate payment processing
  await new Promise((resolve) => setTimeout(resolve, 3000)); // 3s
}

async function sendConfirmation(email: string): Promise<void> {
  // Simulate email sending
  await new Promise((resolve) => setTimeout(resolve, 1000)); // 1s
}

// Start server
const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`Server listening on port ${PORT}`);
  console.log('Graceful shutdown enabled');
});

/**
 * Kubernetes-ready shutdown
 */
class KubernetesGracefulShutdown extends GracefulShutdown {
  private drainDelay: number;

  constructor(
    server: http.Server,
    resources: ShutdownResource[],
    options: { timeout?: number; drainDelay?: number } = {}
  ) {
    super(server, resources, options);
    this.drainDelay = options.drainDelay || 10000; // 10s default
  }

  /**
   * Override shutdown to add drain delay
   */
  private async shutdownWithDrain(signal: string) {
    console.log(`[K8s] Received ${signal}`);

    // Mark as shutting down (health checks fail)
    this['isShuttingDown'] = true;
```

```
    // Wait for load balancer to remove this pod
    console.log(`[K8s] Draining for ${this.drainDelay}ms...`);
    await new Promise((resolve) => setTimeout(resolve, this.drainDelay));

    // Continue with normal shutdown
    console.log('[K8s] Drain complete, proceeding with shutdown');
    // Call parent shutdown logic
  }
}

/**
 * Worker process shutdown (for background jobs)
 */
class WorkerShutdown {
  private isShuttingDown = false;
  private activeJobs = 0;

  constructor(private shutdownTimeout: number = 30000) {
    this.setupHandlers();
  }

  private setupHandlers() {
    process.on('SIGTERM', () => this.shutdown('SIGTERM'));
  }

  trackJob(jobId: string) {
    this.activeJobs++;
    console.log(`[Worker] Job ${jobId} started (${this.activeJobs} active)`);

    return () => {
      this.activeJobs--;
      console.log(`[Worker] Job ${jobId} finished (${this.activeJobs} active)`);
    };
  }

  isShutdown(): boolean {
    return this.isShuttingDown;
  }

  private async shutdown(signal: string) {
    if (this.isShuttingDown) return;

    console.log(`[Worker] Received ${signal}, stopping job processing...`);
    this.isShuttingDown = true;

    const startTime = Date.now();
    const forceExitTimer = setTimeout(() => {
      console.error('[Worker] Timeout exceeded, forcing exit');
      process.exit(1);
    }, this.shutdownTimeout);
```

```
    // Wait for active jobs
    while (this.activeJobs > 0) {
      const elapsed = Date.now() - startTime;
      console.log(
        `[Worker] Waiting for ${this.activeJobs} jobs to finish (${elapsed}ms
elapsed)...`
      );
      await new Promise((resolve) => setTimeout(resolve, 1000));
    }

    clearTimeout(forceExitTimer);
    console.log('[Worker] All jobs completed, exiting');
    process.exit(0);
  }
}

/**
 * Example worker loop
 */
const workerShutdown = new WorkerShutdown(60000);

async function workerLoop() {
  while (!workerShutdown.isShutdown()) {
    const job = await getNextJob();

    if (!job) {
      await new Promise((resolve) => setTimeout(resolve, 1000));
      continue;
    }

    const finish = workerShutdown.trackJob(job.id);

    try {
      await processJob(job);
    } finally {
      finish();
    }
  }

  console.log('[Worker] Loop exited, no more jobs will be processed');
}

async function getNextJob(): Promise<any> {
  // Fetch next job from queue
  return null;
}

async function processJob(job: any): Promise<void> {
  // Process job
  await new Promise((resolve) => setTimeout(resolve, 5000));
}
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Zero-Downtime Deployments

```
// Load balancer removes instance via /health check
// Server completes active requests
// New instance starts
// Load balancer adds new instance
// Old instance exits cleanly
```

Pattern 2: Rolling Restarts (Kubernetes)

```yaml
spec:
  terminationGracePeriodSeconds: 30
  containers:
    - name: api
      livenessProbe:
        httpGet:
          path: /healthz/liveness
      readinessProbe:
        httpGet:
          path: /healthz/readiness
```

Pattern 3: Background Job Processing

```
// Stop accepting new jobs
// Wait for active jobs to finish
// Exit cleanly
```

7. Failure Modes & Edge Cases

Stuck Request

Problem: Request hangs, never completes.

Mitigation: Force shutdown after timeout (30s).

Database Connection Leak

Problem: Connections not closed, pool exhausted.

Mitigation: Close pool after requests finish.

Load Balancer Delay

Problem: LB keeps sending requests after health check fails.

Mitigation: Add drain delay (10s) before shutdown.

8. Performance Characteristics & Tradeoffs

Shutdown Time

- **Immediate exit:** 0ms (but breaks requests)
- **Graceful shutdown:** 5-10s (depends on active requests)
- **With timeout:** Max 30s

Deployment Speed

- **No graceful shutdown:** Fast but broken
- **With graceful shutdown:** Slower but reliable

9. Foot-Guns & Common Mistakes (DO NOT SKIP)

Mistake 1: No Timeout

Fix: Always set timeout (30s), force exit if exceeded.

Mistake 2: Close DB Before Requests Finish

Fix: Wait for requests, THEN close DB.

Mistake 3: No Health Check Coordination

Fix: Return 503 from /health during shutdown.

Mistake 4: Ignore SIGTERM

Fix: Always handle SIGTERM (K8s sends it).

Mistake 5: No Drain Delay

Fix: Wait 10s after marking unhealthy (LB propagation).

10. When NOT to Use This (Anti-Patterns)

Serverless Functions

AWS Lambda handles shutdown automatically (don't need this).

Stateless Workers

If truly stateless (no in-flight work), immediate exit is fine.

11. Related Concepts (With Contrast)

Circuit Breaker

Difference: Circuit breaker prevents requests, graceful shutdown finishes requests.

Rate Limiting

Difference: Rate limiting controls traffic, graceful shutdown handles shutdown.

Health Checks

Related: Health checks coordinate with graceful shutdown.

12. Production Readiness Checklist

Signal Handling

- [ ] Handle SIGTERM (Kubernetes default)
- [ ] Handle SIGINT (Ctrl+C in terminal)
- [ ] Set termination grace period (30s)
- [ ] Force exit after timeout

Request Tracking

- [ ] Track active request count
- [ ] Stop accepting new requests during shutdown
- [ ] Wait for active requests to complete
- [ ] Return 503 for new requests during shutdown

Resource Cleanup

- [ ] Close database connections
- [ ] Close Redis connections
- [ ] Close message queue connections
- [ ] Flush logs

Health Checks

- [ ] /health returns 503 during shutdown
- [ ] /healthz/liveness always returns 200
- [ ] /healthz/readiness returns 503 during shutdown
- [ ] Load balancer uses readiness probe

Monitoring

- [ ] Log shutdown events
- [ ] Track shutdown duration
- [ ] Alert if shutdown timeout exceeded
- [ ] Dashboard: active requests during shutdown