MUSCLEMAP

Enterprise Architecture Refactoring Guide

Modular Design Patterns • Performance Optimization • Scalability

Version 2.0 | December 2024

Jean-Paul Niko | triptomean.com

Executive Summary

This document outlines a comprehensive refactoring strategy for MuscleMap's codebase, transforming it from a monolithic Express.js application into a modular, class-based architecture designed for maximum throughput, minimal resource usage, and unlimited horizontal scalability. The approach draws from battle-tested patterns used at companies like Netflix, Uber, and Stripe.

Key Objectives:

- Eliminate race conditions through immutable state and event sourcing
- Maximize concurrent user capacity with connection pooling and async I/O
- Enable feature additions through plugin architecture and dependency injection
- Create surgical debugging through comprehensive observability

Current State Analysis

Based on our development history, MuscleMap's current architecture presents several challenges that this refactoring addresses:

Identified Issues

1. **Route Registration Order:** Routes placed after app.listen() never register, causing 405 errors

2. **Circular Dependencies:** "Cannot access 'db' before initialization" errors from import ordering
3. **Missing Middleware Imports:** authenticateToken undefined in route files
4. **Monolithic Route Files:** Single files handling too many concerns (auth, validation, business logic, persistence)
5. **SQLite Write Contention:** Single writer at a time limits concurrent workout logging

Proposed Architecture: Hexagonal + CQRS

The refactored architecture combines Hexagonal Architecture (Ports & Adapters) with Command Query Responsibility Segregation (CQRS) to create a system that's both modular and highly performant.

Core Principles

- **Dependency Inversion:** Core business logic has no knowledge of databases, HTTP, or external services
- **Separation of Reads/Writes:** Different models optimized for queries vs. commands
- **Event-Driven:** State changes emit events, enabling audit logs, analytics, and eventual consistency
- **Class-Based Modules:** TypeScript classes with explicit interfaces, like C++ but with JavaScript flexibility

New Directory Structure

The refactored codebase follows a strict layered architecture where dependencies only flow inward:

```
/var/www/musclemap.me/
├── src/
│   ├── core/                    # Domain layer (zero dependencies)
│   │   ├── entities/            # Domain objects
│   │   │   ├── User.ts
│   │   │   ├── Workout.ts
│   │   │   ├── Exercise.ts
│   │   │   ├── Muscle.ts
│   │   │   └── TrainingUnit.ts
│   │   ├── value-objects/       # Immutable values
│   │   │   ├── CreditBalance.ts
│   │   │   ├── BiasWeight.ts
│   │   │   └── ActivationLevel.ts
│   │   ├── ports/               # Interface definitions
│   │   │   ├── IUserRepository.ts
│   │   │   ├── IWorkoutRepository.ts
│   │   │   ├── ICreditService.ts
│   │   │   └── IEventBus.ts
│   │   └── services/            # Domain services
│   │       ├── TrainingUnitCalculator.ts
│   │       ├── PrescriptionEngine.ts
│   │       └── BiasWeightNormalizer.ts
│   ├── application/             # Use cases (orchestration)
│   │   ├── commands/            # Write operations
│   │   │   ├── LogWorkoutCommand.ts
│   │   │   ├── PurchaseCreditsCommand.ts
│   │   │   └── SwitchArchetypeCommand.ts
│   │   ├── queries/             # Read operations
│   │   │   ├── GetUserProgressQuery.ts
│   │   │   ├── GetPrescriptionQuery.ts
│   │   │   └── GetMuscleActivationQuery.ts
│   │   └── handlers/            # Command/Query handlers
│   ├── infrastructure/          # External adapters
│   │   ├── persistence/
│   │   │   ├── SQLiteUserRepository.ts
│   │   │   ├── PostgresUserRepository.ts
│   │   │   └── ConnectionPool.ts
│   │   ├── http/
│   │   │   ├── ExpressServer.ts
│   │   │   ├── middleware/
│   │   │   └── routes/
│   │   ├── events/
│   │   │   ├── InMemoryEventBus.ts
│   │   │   └── RedisEventBus.ts
│   │   └── external/
│   │       ├── StripeAdapter.ts
│   │       └── HealthKitAdapter.ts
│   └── shared/                  # Cross-cutting concerns
│       ├── logging/
│       ├── errors/
│       └── utils/
├── config/                      # Environment configs
├── tests/                       # Mirror of src/
└── scripts/                     # Build, deploy, migrate
```

Class-Based Design Patterns

Following your request for C++-style class patterns, here are the core abstractions:

1. Entity Base Class

All domain objects inherit from a base Entity class that provides identity, equality, and validation:

```
// src/core/entities/Entity.ts
export abstract class Entity<TId> {
  protected readonly _id: TId;
  protected readonly _createdAt: Date;
  private _domainEvents: DomainEvent[] = [];

  constructor(id: TId) {
    this._id = id;
    this._createdAt = new Date();
  }

  get id(): TId { return this._id; }

  equals(other: Entity<TId>): boolean {
    return this._id === other._id;
  }

  protected addDomainEvent(event: DomainEvent): void {
    this._domainEvents.push(event);
  }

  pullDomainEvents(): DomainEvent[] {
    const events = [...this._domainEvents];
    this._domainEvents = [];
    return events;
  }

  abstract validate(): ValidationResult;
}
```

2. Value Object Pattern

Immutable objects that are compared by value, not identity. Perfect for things like BiasWeight and CreditBalance:

```
// src/core/value-objects/ValueObject.ts
export abstract class ValueObject<T> {
  protected readonly props: T;

  constructor(props: T) {
    this.props = Object.freeze(props);
  }
  equals(vo: ValueObject<T>): boolean {
    return JSON.stringify(this.props) === JSON.stringify(vo.props);
  }
}

// src/core/value-objects/BiasWeight.ts
export class BiasWeight extends ValueObject<{ value: number }> {
  private constructor(value: number) {
    super({ value });
  }

  static create(value: number): Result<BiasWeight> {
    if (value <= 0 || value > 10) {
      return Result.fail('BiasWeight must be between 0 and 10');
    }
    return Result.ok(new BiasWeight(value));
  }

  normalize(rawActivation: number): number {
    return rawActivation / this.props.value;
  }
}
```

3. Repository Interface Pattern

Abstract interface that hides storage details. Enables swapping SQLite for PostgreSQL without changing business logic:

```
// src/core/ports/IWorkoutRepository.ts
export interface IWorkoutRepository {
  findById(id: WorkoutId): Promise<Workout | null>;
  findByUser(userId: UserId, options?: QueryOptions): Promise<Workout[]>;
  save(workout: Workout): Promise<void>;
  delete(id: WorkoutId): Promise<void>;
  getActivationSummary(userId: UserId, period: DateRange):
Promise<MuscleActivationMap>;
}
```

4. Command/Query Handlers

Separate write operations (commands) from read operations (queries) for better performance and maintainability:

```
// src/application/commands/LogWorkoutCommand.ts
export class LogWorkoutCommand {
  constructor(
    public readonly userId: string,
    public readonly exercises: ExerciseInput[],
    public readonly notes?: string
  ) {}
}

// src/application/handlers/LogWorkoutHandler.ts
export class LogWorkoutHandler implements
ICommandHandler<LogWorkoutCommand> {
  constructor(
    private readonly workoutRepo: IWorkoutRepository,
    private readonly creditService: ICreditService,
    private readonly tuCalculator: TrainingUnitCalculator,
    private readonly eventBus: IEventBus
  ) {}

  async execute(cmd: LogWorkoutCommand): Promise<Result<WorkoutId>> {
    // 1. Validate credits
    const creditCheck = await this.creditService.canDebit(cmd.userId, 25);
    if (!creditCheck.success) return Result.fail('Insufficient credits');

    // 2. Calculate TUs
    const activations = this.tuCalculator.calculate(cmd.exercises);

    // 3. Create workout entity
    const workout = Workout.create(cmd.userId, activations, cmd.notes);

    // 4. Persist and debit (transactional)
    await this.workoutRepo.save(workout);
    await this.creditService.debit(cmd.userId, 25);

    // 5. Publish events
    await this.eventBus.publishAll(workout.pullDomainEvents());

    return Result.ok(workout.id);
  }
}
```

Dependency Injection Container

A lightweight DI container wires everything together at startup, making it trivial to swap implementations or mock for testing:

```
// src/shared/Container.ts
export class Container {
  private static registry = new Map<string, any>();
  private static singletons = new Map<string, any>();

  static register<T>(token: string, factory: () => T, singleton = true): void {
    this.registry.set(token, { factory, singleton });
  }

  static resolve<T>(token: string): T {
    const entry = this.registry.get(token);
    if (!entry) throw new Error(`No registration for ${token}`);

    if (entry.singleton) {
      if (!this.singletons.has(token)) {
        this.singletons.set(token, entry.factory());
      }
      return this.singletons.get(token);
    }
    return entry.factory();
  }
}

// Bootstrap (src/bootstrap.ts)
Container.register('IWorkoutRepository', () => new SQLiteWorkoutRepository());
Container.register('ICreditService', () => new CreditService());
Container.register('LogWorkoutHandler', () => new LogWorkoutHandler(
  Container.resolve('IWorkoutRepository'),
  Container.resolve('ICreditService'),
  Container.resolve('TrainingUnitCalculator'),
  Container.resolve('IEventBus')
));
```

Race Condition Prevention

The current codebase has potential race conditions in credit deduction and workout logging. Here's how we eliminate them:

1. Optimistic Locking

Add a version column to detect concurrent modifications:

```
// Database schema addition
ALTER TABLE credit_wallets ADD COLUMN version INTEGER DEFAULT 0;

// Repository method
async debit(userId: string, amount: number): Promise<Result<void>> {
  const wallet = await this.findByUserId(userId);

  const result = await db.run(`
    UPDATE credit_wallets
    SET balance = balance - ?, version = version + 1
    WHERE user_id = ? AND version = ? AND balance >= ?
  `, [amount, userId, wallet.version, amount]);

  if (result.changes === 0) {
    return Result.fail('Concurrent modification or insufficient funds');
  }
  return Result.ok();
}
```

2. Idempotency Keys

Prevent duplicate operations from network retries:

```
// Middleware for idempotent operations
export class IdempotencyMiddleware {
  private cache = new Map<string, { result: any; expiry: number }>();

  async wrap<T>(key: string, operation: () => Promise<T>): Promise<T> {
    const cached = this.cache.get(key);
    if (cached && cached.expiry > Date.now()) {
      return cached.result;
    }

    const result = await operation();
    this.cache.set(key, { result, expiry: Date.now() + 3600000 });
    return result;
  }
}
```

3. SQLite WAL Mode with Retry

For SQLite, enable WAL mode and implement smart retries:

```
// src/infrastructure/persistence/SQLiteConnection.ts
export class SQLiteConnection {
  private db: Database;

  constructor(path: string) {
    this.db = new Database(path);
    this.db.pragma('journal_mode = WAL');
    this.db.pragma('synchronous = NORMAL');
    this.db.pragma('cache_size = -64000'); // 64MB
    this.db.pragma('busy_timeout = 5000');  // 5s retry
  }

  async transaction<T>(fn: (tx: Transaction) => Promise<T>): Promise<T> {
    return this.db.transaction(fn)();
  }
}
```

Performance Optimization Strategy

1. Read/Write Separation

Use separate optimized models for reads vs. writes. Writes go through domain entities; reads bypass them for speed:

```
// Read model - optimized for dashboard queries
CREATE VIEW user_dashboard_view AS
SELECT
  u.id, u.username, u.credit_balance,
  COUNT(DISTINCT w.id) as workout_count,
  SUM(w.total_tu) as total_tu,
  a.name as current_archetype,
  al.level as current_level
FROM users u
LEFT JOIN workouts w ON w.user_id = u.id
LEFT JOIN archetypes a ON a.id = u.current_archetype_id
LEFT JOIN archetype_levels al ON al.id = u.current_level_id
GROUP BY u.id;
```

2. Connection Pooling

When you scale to PostgreSQL, proper connection pooling prevents resource exhaustion:

```
// src/infrastructure/persistence/ConnectionPool.ts
import { Pool, PoolConfig } from 'pg';

export class ConnectionPool {
  private pool: Pool;

  constructor() {
    this.pool = new Pool({
      max: 20,                    // Max connections
      idleTimeoutMillis: 30000,   // Close idle after 30s
      connectionTimeoutMillis: 2000 // Fail fast
    });
  }

  async query<T>(sql: string, params?: any[]): Promise<T[]> {
    const client = await this.pool.connect();
    try {
      const result = await client.query(sql, params);
      return result.rows;
    } finally {
      client.release();
    }
  }
}
```

3. Response Caching

Cache expensive computations like muscle activation maps:

```
// src/infrastructure/cache/ActivationCache.ts
export class ActivationCache {
  private cache = new Map<string, { data: any; expiry: number }>();
  private readonly TTL = 60000; // 1 minute

  async get<T>(key: string, factory: () => Promise<T>): Promise<T> {
    const cached = this.cache.get(key);
    if (cached && cached.expiry > Date.now()) {
      return cached.data as T;
    }

    const data = await factory();
    this.cache.set(key, { data, expiry: Date.now() + this.TTL });
    return data;
  }

  invalidate(pattern: string): void {
    for (const key of this.cache.keys()) {
      if (key.includes(pattern)) this.cache.delete(key);
    }
  }
}
```

Enhanced Logging & Observability

Building on your existing logging system, here's a structured approach for surgical debugging:

Structured Logger

```
// src/shared/logging/Logger.ts
export class Logger {
  private context: Record<string, any> = {};

  constructor(private name: string) {}

  child(context: Record<string, any>): Logger {
    const child = new Logger(this.name);
    child.context = { ...this.context, ...context };
    return child;
  }

  info(message: string, data?: Record<string, any>): void {
    this.log('INFO', message, data);
  }

  error(message: string, error?: Error, data?: Record<string, any>): void {
    this.log('ERROR', message, {
      ...data,
      error: error ? { message: error.message, stack: error.stack } :
undefined
    });
  }

  private log(level: string, message: string, data?: Record<string, any>):
void {
    console.log(JSON.stringify({
      timestamp: new Date().toISOString(),
      level,
      logger: this.name,
      message,
      ...this.context,
      ...data
    }));
  }
}
```

Request Tracing Middleware

```
// src/infrastructure/http/middleware/tracing.ts
export const tracingMiddleware = (req: Request, res: Response, next: 
NextFunction) => {
  const requestId = req.headers['x-request-id'] || crypto.randomUUID();
  const startTime = Date.now();

  req.logger = logger.child({ requestId, userId: req.user?.id });
  res.setHeader('x-request-id', requestId);

  res.on('finish', () => {
    req.logger.info('Request completed', {
      method: req.method,
      path: req.path,
      status: res.statusCode,
      durationMs: Date.now() - startTime
    });
  });

  next();
};
```

Plugin Architecture for Feature Expansion

To enable easy feature additions, implement a plugin system that hooks into the application lifecycle:

```
// src/shared/plugins/Plugin.ts
export interface IPlugin {
  name: string;
  version: string;
  initialize(container: Container): Promise<void>;
  registerRoutes?(router: Router): void;
  registerEventHandlers?(eventBus: IEventBus): void;
  shutdown?(): Promise<void>;
}

// Example: Garmin Integration Plugin
export class GarminPlugin implements IPlugin {
  name = 'garmin-integration';
  version = '1.0.0';

  async initialize(container: Container): Promise<void> {
    container.register('GarminAdapter', () => new GarminAdapter());
  }

  registerRoutes(router: Router): void {
    router.post('/api/integrations/garmin/sync', this.syncHandler);
    router.get('/api/integrations/garmin/status', this.statusHandler);
  }

  registerEventHandlers(eventBus: IEventBus): void {
    eventBus.subscribe('WorkoutCompleted', this.onWorkoutCompleted);
  }
}
```

Migration Strategy

Refactoring a production app requires a careful, incremental approach. Here's the recommended sequence:

Phase 1: Foundation (Week 1-2)

1. Create new directory structure alongside existing code
2. Implement base classes: Entity, ValueObject, Result
3. Set up dependency injection container
4. Add structured logging middleware

Phase 2: Core Domain (Week 3-4)

1. Extract domain entities: User, Workout, Exercise, Muscle
2. Create value objects: BiasWeight, CreditBalance, ActivationLevel
3. Define repository interfaces
4. Migrate TrainingUnitCalculator to domain service

Phase 3: Application Layer (Week 5-6)

1. Create commands: LogWorkout, PurchaseCredits, SwitchArchetype
2. Create queries: GetUserProgress, GetPrescription, GetActivation
3. Implement handlers with proper error handling
4. Add optimistic locking to credit operations

Phase 4: Infrastructure (Week 7-8)

1. Implement SQLite repository adapters
2. Refactor routes to use new handlers
3. Add request tracing and metrics
4. Implement caching layer
5. Set up plugin loading system

Scaling Path

With this architecture, here's how you scale as usage grows:

Users	Database	App Servers	Cache
0-10K	SQLite + WAL	Single Node.js	In-memory
10K-100K	PostgreSQL	2-4 Node.js + LB	Redis
100K-1M	PG + Read Replicas	Auto-scaling cluster	Redis Cluster
1M+	Sharded PG +	Kubernetes	Multi-tier caching

Conclusion

This architecture positions MuscleMap for growth from hundreds to millions of users without requiring major rewrites. The modular design means you can add Garmin integration, Apple Vision Pro support, or real-time wearable sync as plugins without touching core business logic.

The class-based approach with dependency injection makes the codebase feel like a well-structured C++ project while retaining JavaScript's flexibility. Most importantly, the separation of concerns makes bugs easy to isolate and features easy to add.

Next step: Start with Phase 1 - create the foundation classes and logging infrastructure. This gives immediate debugging benefits while setting up for the full migration.