TaskoMask Solution Architecture Documentation

Table of Contents

1. Introduction

TaskoMask is a sophisticated task management system built using modern microservices architecture. The solution implements industry best practices including CQRS (Command Query Responsibility Segregation), Event Sourcing, and Domain-Driven Design.

1.1 Purpose

The system provides a scalable and maintainable platform for:

- Task and project management
- Team collaboration
- Process tracking
- Resource organization

1.2 Key Features

- Microservices-based architecture
- Event-driven design
- Scalable data management
- Secure authentication and authorization
- Real-time updates

2. System Overview

2.1 Core Services

1. **Boards Service**
    - Manages project boards and cards
    - Implements CQRS pattern
    - Handles board-related operations
2. **Tasks Service**
    - Task management and tracking
    - Comment handling
    - Status updates
3. **Identity Service**
    - User authentication and authorization
    - Role-based access control

    - OAuth2/OpenID Connect implementation

4. **Owners Service**
    - Permission management
    - Resource ownership
    - Access control

2.2 Supporting Infrastructure

- API Gateway
- Message Queue (RabbitMQ)
- Event Store (Redis)
- Document Database (MongoDB)

3. Architecture

3.1 Architectural Patterns

CQRS Implementation

The solution implements CQRS through:

- Separate Read and Write APIs
- Dedicated read and write databases
- Event-driven synchronization

```
Command Flow:
Client → API Gateway → Write API → Event Store → Read Model Update

Query Flow:
Client → API Gateway → Read API → Optimized Read Store
```

Event Sourcing

- Events as source of truth
- Redis-based event store
- Event replay capability
- Audit trail support

Microservices Communication

- Event-driven using MassTransit/RabbitMQ
- gRPC for synchronous operations
- REST APIs for client communication

3.2 Building Blocks

The solution's foundation is built on shared components:

1. **Domain Layer**
    - Core business logic
    - Entity definitions
    - Value objects
    - Domain events

2. **Application Layer**
    - Use case implementations
    - Command/Query handlers
    - Application services
3. **Infrastructure Layer**
    - Technical implementations
    - External service integrations
    - Data persistence
4. **Contracts**
    - DTOs
    - Event definitions
    - API contracts

4. Core Components

4.1 Event Store Implementation

```
public class RedisEventStoreService : IEventStoreService
{
    private readonly IConnectionMultiplexer _redisConnection;
    private readonly IDatabase _redisDb;

    public async Task SaveAsync<TDomainEvent>(TDomainEvent @event)
        where TDomainEvent : DomainEvent
    {
        var storedEvent = GetEventDataToStore(@event);
        await _redisDb.ListLeftPushAsync(MakeKey(@event.EntityId), jsonData);
    }
}
```

4.2 Message Queue Integration

```
public class MassTransitEventPublisher : IEventPublisher
{
    private readonly IPublishEndpoint _publishEndpoint;

    public async Task Publish<TEvent>(TEvent @event)
        where TEvent : IIntegrationEvent
    {
        await _publishEndpoint.Publish(@event);
    }
}
```

4.3 API Gateway

- Route aggregation
- Authentication middleware

- Request transformation
- Load balancing

5. Technical Implementation

5.1 Service Implementation

Each service follows Clean Architecture:

```
ServiceName/
├── Domain/          # Business logic
├── Application/     # Use cases
├── Infrastructure/  # Technical details
└── API/            # Controllers
```

5.2 Data Management

- Event sourcing for write operations
- MongoDB for read models
- Redis for caching
- Event store for audit trails

5.3 Security

- JWT-based authentication
- OAuth2 authorization
- Role-based access control
- Scope-based permissions

6. Deployment

6.1 Container Support

- Docker containers for each service
- Docker Compose for development
- Kubernetes-ready configuration

6.2 Configuration Management

- Environment-specific settings
- Secret management
- Feature toggles

7. Best Practices

7.1 Development Guidelines

- Clean Architecture principles
- Domain-Driven Design
- SOLID principles
- Event-driven design

7.2 Testing Strategy

- Unit tests

- Integration tests
- Event sourcing tests
- API tests

7.3 Monitoring and Logging

- OpenTelemetry integration
- Centralized logging
- Metrics collection
- Distributed tracing

Conclusion

TaskoMask demonstrates a modern approach to building scalable, maintainable microservices. Its implementation of CQRS, event sourcing, and clean architecture provides a robust foundation for complex business applications.

Detailed Architecture

Service Communication Patterns

1. Synchronous Communication

TaskoMask uses gRPC for efficient service-to-service communication where immediate response is required:

```
public class GetBoardByIdHandler : IRequestHandler<GetBoardByIdRequest,
BoardDetailsViewModel>
{
    private readonly GetBoardByIdGrpcServiceClient _getBoardByIdGrpcServiceClient;
    private readonly GetCardsByBoardIdGrpcServiceClient _getCardsByBoardIdGrpcServiceClient;

    public async Task<BoardDetailsViewModel> Handle(GetBoardByIdRequest request)
    {
        var board = await GetBoardAsync(request.Id);
        var cards = await GetCardsAsync(request.Id);
        return new BoardDetailsViewModel { Board = board, Cards = cards };
    }
}
```

2. Asynchronous Communication

Event-driven communication using MassTransit and RabbitMQ:

```
public class TaskStatusUpdatedConsumer : IConsumer<TaskStatusUpdated>
{
    private readonly ITaskReadModelRepository _repository;

    public async Task Consume(ConsumeContext<TaskStatusUpdated> context)
    {
        var @event = context.Message;
        await _repository.UpdateTaskStatus(@event.TaskId, @event.NewStatus);
```

```
    }
}
```

Domain Model Design

1. Aggregate Roots

Example of Task aggregate:

```
public class Task : Entity, IAggregateRoot
{
    private readonly List<Comment> _comments;

    public string Title { get; private set; }
    public TaskStatus Status { get; private set; }
    public string AssigneeId { get; private set; }

    public void UpdateStatus(TaskStatus newStatus)
    {
        Status = newStatus;
        AddDomainEvent(new TaskStatusUpdated(Id, newStatus));
    }

    public void AddComment(string content, string userId)
    {
        var comment = new Comment(content, userId);
        _comments.Add(comment);
        AddDomainEvent(new CommentAdded(Id, comment.Id));
    }
}
```

2. Value Objects

Example of immutable value objects:

```
public class TaskStatus : ValueObject
{
    public string Value { get; }

    private TaskStatus(string value)
    {
        Value = value;
    }

    public static TaskStatus ToDo = new TaskStatus("ToDo");
    public static TaskStatus InProgress = new TaskStatus("InProgress");
    public static TaskStatus Done = new TaskStatus("Done");

    protected override IEnumerable<object> GetEqualityComponents()
    {
        yield return Value;
```

```
    }
}
```

Event Sourcing Implementation

1. Event Store

Redis-based event store implementation:

```
public class RedisEventStoreService : IEventStoreService
{
    private readonly IDatabase _redisDb;

    public async Task SaveAsync<TEvent>(TEvent @event) where TEvent : IDomainEvent
    {
        var storedEvent = new StoredEvent
        {
            Id = Guid.NewGuid().ToString(),
            EntityId = @event.EntityId,
            EntityType = @event.EntityType,
            EventType = @event.GetType().Name,
            Data = JsonConvert.SerializeObject(@event),
            Timestamp = DateTime.UtcNow
        };

        await _redisDb.ListLeftPushAsync(
            $"events:{@event.EntityType}:{@event.EntityId}",
            JsonConvert.SerializeObject(storedEvent)
        );
    }
}
```

2. Event Replay

Capability to rebuild state from events:

```
public class TaskEventRebuilder
{
    private readonly IEventStoreService _eventStore;

    public async Task<Task> RebuildTaskState(string taskId)
    {
        var events = await _eventStore.GetEventsAsync("Task", taskId);
        var task = new Task(); // Create empty state

        foreach (var @event in events.OrderBy(e => e.Timestamp))
        {
            task.Apply(@event); // Apply each event in sequence
        }

        return task;
```

```
    }
}
```

Security Implementation

1. Authentication

JWT-based authentication with Identity Service:

```
public class AuthenticationConfig
{
    public void Configure(IServiceCollection services, IConfiguration configuration)
    {
        services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
            .AddJwtBearer(options =>
            {
                options.Authority = configuration["Jwt:Authority"];
                options.Audience = configuration["Jwt:Audience"];
                options.RequireHttpsMetadata = false;
            });
    }
}
```

2. Authorization

Fine-grained permission control:

```
[Authorize("task-write-access")]
public class UpdateTaskStatusEndpoint : EndpointBase
{
    [HttpPut("tasks/{id}/status")]
    public async Task<IActionResult> UpdateStatus(
        string id,
        [FromBody] UpdateTaskStatusRequest request)
    {
        if (!await _authorizationService.CanModifyTask(User, id))
            return Forbid();

        var command = new UpdateTaskStatusCommand(id, request.Status);
        await _mediator.Send(command);
        return Ok();
    }
}
```

Monitoring and Telemetry

1. Distributed Tracing

OpenTelemetry integration:

```
public static class OpenTelemetryExtensions
{
    public static void AddOpenTelemetry(this IServiceCollection services, IConfiguration
config)
    {
        services.AddOpenTelemetryTracing(builder =>
        {
            builder
                .SetResourceBuilder(ResourceBuilder
                    .CreateDefault()
                    .AddService(config["OpenTelemetry:ServiceName"]))
                .AddAspNetCoreInstrumentation()
                .AddHttpClientInstrumentation()
                .AddGrpcClientInstrumentation()
                .AddRedisInstrumentation()
                .AddMassTransitInstrumentation()
                .AddOtlpExporter(opts =>
                {
                    opts.Endpoint = new Uri(config["OpenTelemetry:Endpoint"]);
                });
        });
    }
}
```

2. Metrics Collection

Key metrics monitoring:

```
public class MetricsCollector
{
    private readonly Meter _meter;
    private readonly Counter<long> _taskCreatedCounter;
    private readonly Histogram<double> _taskCompletionTime;

    public MetricsCollector()
    {
        _meter = new Meter("TaskoMask.Tasks");
        _taskCreatedCounter = _meter.CreateCounter<long>("tasks_created_total");
        _taskCompletionTime = _meter.CreateHistogram<double>("task_completion_seconds");
    }

    public void RecordTaskCreated()
    {
        _taskCreatedCounter.Add(1);
    }

    public void RecordTaskCompletion(TimeSpan duration)
    {
        _taskCompletionTime.Record(duration.TotalSeconds);
```

```
    }
}
```

Deployment Guide

Infrastructure Requirements

1. Core Services

- .NET 6.0 Runtime
- Redis 6.x or higher
- RabbitMQ 3.8 or higher
- MongoDB 4.4 or higher

2. Development Tools

- Docker Desktop
- .NET SDK 6.0
- Visual Studio 2022 or VS Code

Docker Deployment

1. Service Containerization

Example Dockerfile for Tasks.Write.Api:

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["src/2-Services/Tasks/Api/Tasks.Write.Api/Tasks.Write.Api.csproj", "Tasks.Write.Api/"]
RUN dotnet restore "Tasks.Write.Api/Tasks.Write.Api.csproj"
COPY . .
WORKDIR "/src/Tasks.Write.Api"
RUN dotnet build "Tasks.Write.Api.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "Tasks.Write.Api.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Tasks.Write.Api.dll"]
```

2. Docker Compose Configuration

```yaml
version: '3.8'

services:
  redis:
    image: redis:6.2-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis-data:/data
    command: redis-server --appendonly yes

  rabbitmq:
    image: rabbitmq:3.8-management-alpine
    ports:
      - "5672:5672"
      - "15672:15672"
    volumes:
      - rabbitmq-data:/var/lib/rabbitmq
    environment:
      - RABBITMQ_DEFAULT_USER=taskomask
      - RABBITMQ_DEFAULT_PASS=your_secure_password

  mongodb:
    image: mongo:4.4
    ports:
      - "27017:27017"
    volumes:
      - mongodb-data:/data/db
    environment:
      - MONGO_INITDB_ROOT_USERNAME=taskomask
      - MONGO_INITDB_ROOT_PASSWORD=your_secure_password

  identity-api:
    build:
      context: .
      dockerfile: src/2-Services/Identity/Api/Identity.Api/Dockerfile
    ports:
      - "5001:80"
    environment:
      - ASPNETCORE_ENVIRONMENT=Production
      - ConnectionStrings__Redis=redis:6379
      - RabbitMQ__Host=rabbitmq

  tasks-write-api:
    build:
      context: .
      dockerfile: src/2-Services/Tasks/Api/Tasks.Write.Api/Dockerfile
    ports:
      - "5002:80"
    depends_on:
      - redis
```

```
      - rabbitmq
    environment:
      - ASPNETCORE_ENVIRONMENT=Production
      - EventStore__ConnectionString=redis:6379
      - RabbitMQ__Host=rabbitmq

  tasks-read-api:
    build:
      context: .
      dockerfile: src/2-Services/Tasks/Api/Tasks.Read.Api/Dockerfile
    ports:
      - "5003:80"
    depends_on:
      - mongodb
      - rabbitmq
    environment:
      - ASPNETCORE_ENVIRONMENT=Production
      - MongoDB__ConnectionString=mongodb://taskomask:your_secure_password@mongodb:27017
      - RabbitMQ__Host=rabbitmq

volumes:
  redis-data:
  rabbitmq-data:
  mongodb-data:
```

Kubernetes Deployment

1. Service Configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tasks-write-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: tasks-write-api
  template:
    metadata:
      labels:
        app: tasks-write-api
    spec:
      containers:
      - name: tasks-write-api
        image: taskomask/tasks-write-api:latest
        ports:
        - containerPort: 80
        env:
        - name: ASPNETCORE_ENVIRONMENT
```

```
          value: "Production"
        - name: EventStore__ConnectionString
          valueFrom:
            secretKeyRef:
              name: redis-secret
              key: connection-string
        - name: RabbitMQ__Host
          valueFrom:
            configMapKeyRef:
              name: rabbitmq-config
              key: host
        resources:
          requests:
            memory: "128Mi"
            cpu: "100m"
          limits:
            memory: "256Mi"
            cpu: "200m"
```

2. Service Discovery

```
apiVersion: v1
kind: Service
metadata:
  name: tasks-write-api
spec:
  selector:
    app: tasks-write-api
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

3. Ingress Configuration

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: taskomask-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: api.taskomask.com
    http:
      paths:
      - path: /tasks/write
        pathType: Prefix
```

```
        backend:
          service:
            name: tasks-write-api
            port:
              number: 80
      - path: /tasks/read
        pathType: Prefix
        backend:
          service:
            name: tasks-read-api
            port:
              number: 80
```

Configuration Management

1. Application Settings

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "EventStore": {
    "ConnectionString": "localhost:6379",
    "Database": 0
  },
  "RabbitMQ": {
    "Host": "localhost",
    "Username": "guest",
    "Password": "guest",
    "VirtualHost": "/"
  },
  "JWT": {
    "Authority": "https://identity.taskomask.com",
    "Audience": "tasks-api",
    "RequireHttpsMetadata": true
  },
  "OpenTelemetry": {
    "ServiceName": "Tasks.Write.Api",
    "Endpoint": "http://otel-collector:4317"
  }
}
```

2. Secret Management

Using Azure Key Vault (example):

```
public static class KeyVaultExtensions
{
    public static void AddAzureKeyVault(this IServiceCollection services, IConfiguration
config)
    {
        var keyVaultUrl = config["KeyVault:Url"];
        var credential = new DefaultAzureCredential();

        services.Configure<AzureKeyVaultConfigurationOptions>(options =>
        {
            options.ReloadInterval = TimeSpan.FromHours(1);
        });

        services.AddAzureKeyVault(new Uri(keyVaultUrl), credential);
    }
}
```

Monitoring Setup

1. Health Checks

```
public static class HealthCheckExtensions
{
    public static void AddHealthChecks(this IServiceCollection services, IConfiguration
config)
    {
        services.AddHealthChecks()
            .AddRedis(config["EventStore:ConnectionString"], name: "redis")
            .AddRabbitMQ(config["RabbitMQ:ConnectionString"], name: "rabbitmq")
            .AddMongoDb(config["MongoDB:ConnectionString"], name: "mongodb");
    }
}
```

2. Logging Configuration

```
public static class LoggingExtensions
{
    public static void ConfigureLogging(this ILoggingBuilder logging)
    {
        logging.ClearProviders();
        logging.AddConsole();
        logging.AddDebug();
        logging.AddApplicationInsights();

        logging.AddFilter("Microsoft", LogLevel.Warning);
        logging.AddFilter("System", LogLevel.Warning);
    }
}
```

Development Guide

Project Structure

```
TaskoMask/
├── src/
│   ├── 1-BuildingBlocks/           # Shared libraries and components
│   │   ├── Domain/                 # Core domain objects
│   │   ├── Application/            # Application services
│   │   ├── Infrastructure/         # Technical implementations
│   │   └── Contracts/              # Shared DTOs and events
│   │
│   ├── 2-Services/                 # Microservices
│   │   ├── Tasks/                  # Task management service
│   │   │   ├── Api/
│   │   │   │   ├── Tasks.Read.Api/
│   │   │   │   └── Tasks.Write.Api/
│   │   │   └── Tests/
│   │   │
│   │   ├── Boards/                 # Board management service
│   │   │   ├── Api/
│   │   │   │   ├── Boards.Read.Api/
│   │   │   │   └── Boards.Write.Api/
│   │   │   └── Tests/
│   │   │
│   │   └── Identity/               # Identity service
│   │       ├── Api/
│   │       └── Tests/
│   │
│   └── 3-ApiGateways/             # API Gateways
│       └── UserPanel/
│
├── tests/                          # Solution-wide tests
│   ├── Integration.Tests/
│   └── Load.Tests/
│
└── tools/                          # Development tools and scripts
```

Development Setup

1. Prerequisites

- .NET 6.0 SDK
- Docker Desktop
- Visual Studio 2022 or VS Code
- Git

2. Initial Setup

```
# Clone repository
git clone https://github.com/yourusername/TaskoMask.git
cd TaskoMask

# Restore dependencies
dotnet restore TaskoMask.sln

# Start infrastructure services
docker-compose -f docker-compose.infrastructure.yml up -d

# Build solution
dotnet build TaskoMask.sln
```

3. Development Workflow

Creating a New Feature

1. Create feature branch:

```
git checkout -b feature/your-feature-name
```

2. Implement the feature following the DDD and Clean Architecture principles:

```
// Domain Entity
public class Task : Entity, IAggregateRoot
{
    public string Title { get; private set; }
    public TaskStatus Status { get; private set; }

    private Task() { } // For EF Core

    public Task(string title)
    {
        Title = title;
        Status = TaskStatus.ToDo;
        AddDomainEvent(new TaskCreatedEvent(this));
    }

    public void UpdateStatus(TaskStatus newStatus)
    {
        Status = newStatus;
        AddDomainEvent(new TaskStatusUpdatedEvent(this));
    }
}

// Application Command
public class CreateTaskCommand : IRequest<Result<string>>
{
    public string Title { get; }
```

```
    public CreateTaskCommand(string title)
    {
        Title = title;
    }
}

// Command Handler
public class CreateTaskCommandHandler
    : IRequestHandler<CreateTaskCommand, Result<string>>
{
    private readonly ITaskRepository _repository;

    public async Task<Result<string>> Handle(
        CreateTaskCommand command,
        CancellationToken cancellationToken)
    {
        var task = new Task(command.Title);
        await _repository.AddAsync(task);
        return Result.Success(task.Id);
    }
}
```

3. Add tests:

```
public class CreateTaskTests
{
    [Fact]
    public async Task Handle_ValidCommand_CreatesTask()
    {
        // Arrange
        var repository = new Mock<ITaskRepository>();
        var handler = new CreateTaskCommandHandler(repository.Object);
        var command = new CreateTaskCommand("Test Task");

        // Act
        var result = await handler.Handle(command, CancellationToken.None);

        // Assert
        Assert.True(result.IsSuccess);
        repository.Verify(r => r.AddAsync(It.IsAny<Task>()), Times.Once);
    }
}
```

Testing Strategy

1. Unit Tests

Focus on testing business logic in isolation:

```
public class TaskTests
{
    [Fact]
    public void UpdateStatus_ChangesStatus_RaisesEvent()
    {
        // Arrange
        var task = new Task("Test Task");

        // Act
        task.UpdateStatus(TaskStatus.InProgress);

        // Assert
        Assert.Equal(TaskStatus.InProgress, task.Status);
        Assert.Contains(task.DomainEvents,
            e => e is TaskStatusUpdatedEvent);
    }
}
```

2. Integration Tests

Test multiple components working together:

```
public class TasksApiIntegrationTests : IClassFixture<WebApplicationFactory<Program>>
{
    private readonly WebApplicationFactory<Program> _factory;

    [Fact]
    public async Task CreateTask_ValidRequest_ReturnsCreated()
    {
        // Arrange
        var client = _factory.CreateClient();
        var request = new CreateTaskRequest { Title = "Test Task" };

        // Act
        var response = await client.PostAsJsonAsync("/api/tasks", request);

        // Assert
        response.EnsureSuccessStatusCode();
        Assert.Equal(HttpStatusCode.Created, response.StatusCode);
    }
}
```

3. Event Sourcing Tests

Verify event handling and replay:

```
public class TaskEventSourceTests
{
    [Fact]
    public async Task ReplayEvents_RestoresTaskState()
```

```
    {
        // Arrange
        var events = new List<IDomainEvent>
        {
            new TaskCreatedEvent("Test Task"),
            new TaskStatusUpdatedEvent(TaskStatus.InProgress),
            new TaskStatusUpdatedEvent(TaskStatus.Done)
        };

        // Act
        var task = await TaskEventRebuilder.Replay(events);

        // Assert
        Assert.Equal(TaskStatus.Done, task.Status);
    }
}
```

Coding Standards

1. Naming Conventions

- Use PascalCase for public members and types
- Use camelCase for private fields
- Prefix interfaces with 'I'
- Use meaningful, descriptive names

2. Code Organization

- One class per file
- Group related files in folders
- Keep classes focused and small
- Follow Clean Architecture layers

3. Error Handling

```
public class ErrorHandling
{
    public async Task<Result<T>> TryOperation<T>(Func<Task<T>> operation)
    {
        try
        {
            var result = await operation();
            return Result.Success(result);
        }
        catch (DomainException ex)
        {
            return Result.Failure<T>(ex.Message);
        }
        catch (Exception ex)
        {
            // Log unexpected error
            _logger.LogError(ex, "Unexpected error occurred");
```

```
            return Result.Failure<T>("An unexpected error occurred");
        }
    }
}
```

4. Documentation

Add XML comments for public APIs:

```
/// <summary>
/// Updates the status of a task and raises appropriate events
/// </summary>
/// <param name="newStatus">The new status to set</param>
/// <returns>Result indicating success or failure</returns>
/// <exception cref="InvalidOperationException">
/// Thrown when status transition is invalid
/// </exception>
public Result UpdateStatus(TaskStatus newStatus)
{
    // Implementation
}
```

Debugging and Troubleshooting

1. Logging

Use structured logging:

```
public class TaskService
{
    private readonly ILogger<TaskService> _logger;

    public async Task<Result> ProcessTask(string taskId)
    {
        _logger.LogInformation(
            "Processing task {TaskId} started at {StartTime}",
            taskId,
            DateTime.UtcNow);

        try
        {
            // Process task
            _logger.LogInformation(
                "Task {TaskId} processed successfully",
                taskId);
            return Result.Success();
        }
        catch (Exception ex)
        {
            _logger.LogError(ex,
                "Error processing task {TaskId}",
```

```
                taskId);
            return Result.Failure(ex.Message);
        }
    }
}
```

2. Debugging Tools

- Use Visual Studio debugger
- Docker container logs
- Application Insights
- OpenTelemetry traces

3. Common Issues

1. Event Store Connection:

```
public async Task<bool> VerifyEventStoreConnection()
{
    try
    {
        await _redisConnection.GetDatabase().PingAsync();
        return true;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Failed to connect to Event Store");
        return false;
    }
}
```

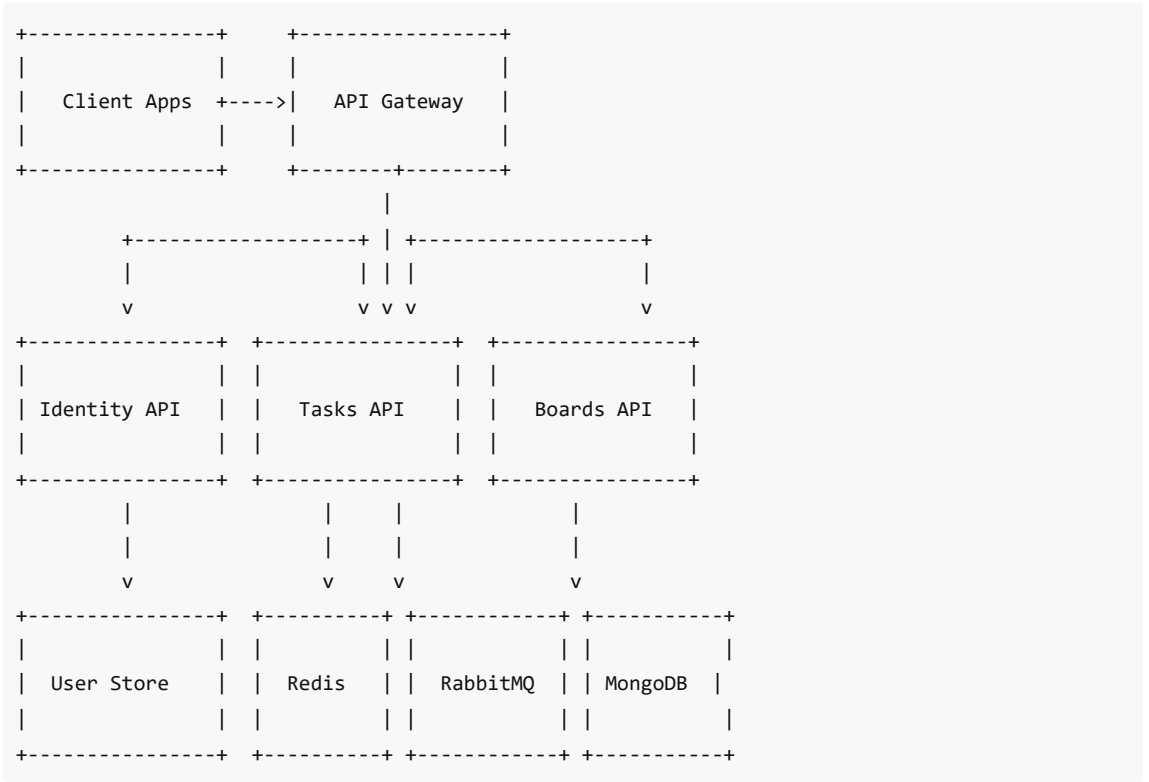
2. Message Queue Issues:

```
public class MessageQueueHealthCheck : IHealthCheck
{
    private readonly IBusControl _bus;

    public async Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default)
    {
        try
        {
            var endpoint = await _bus.GetSendEndpoint(
                new Uri("queue:health-check"));
            return HealthCheckResult.Healthy();
        }
        catch (Exception ex)
        {
            return HealthCheckResult.Unhealthy(ex.Message);
        }
```

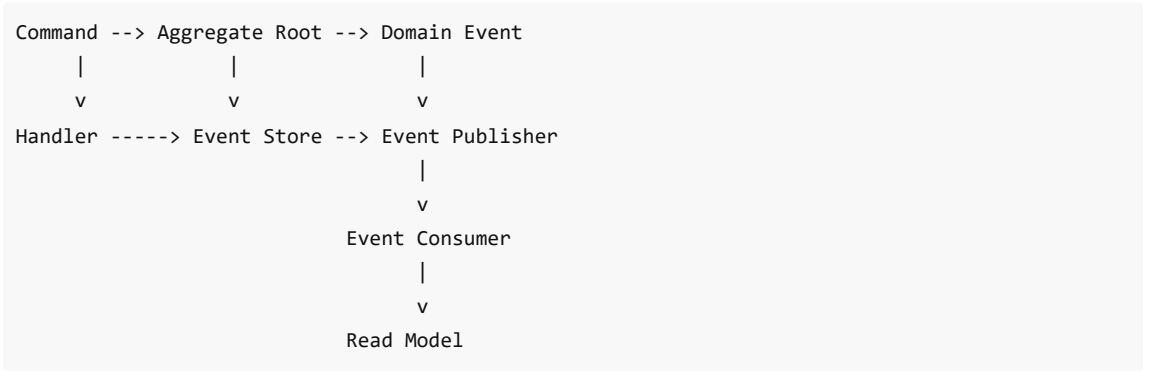
```
    }
}
```

Architecture Diagrams

System Overview



Event Sourcing Flow

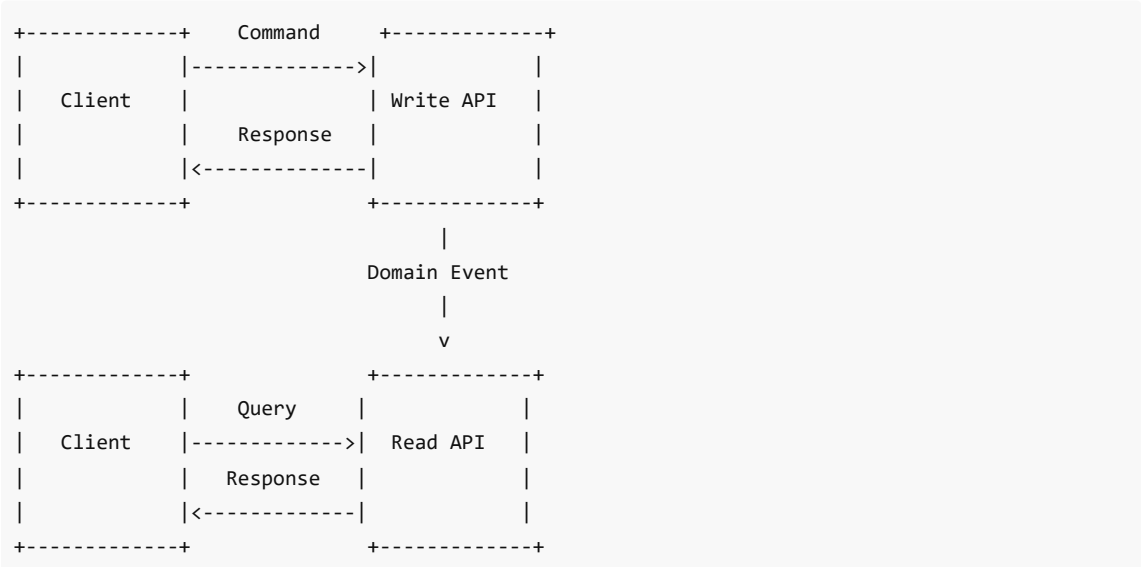


Service Architecture (Example: Tasks Service)

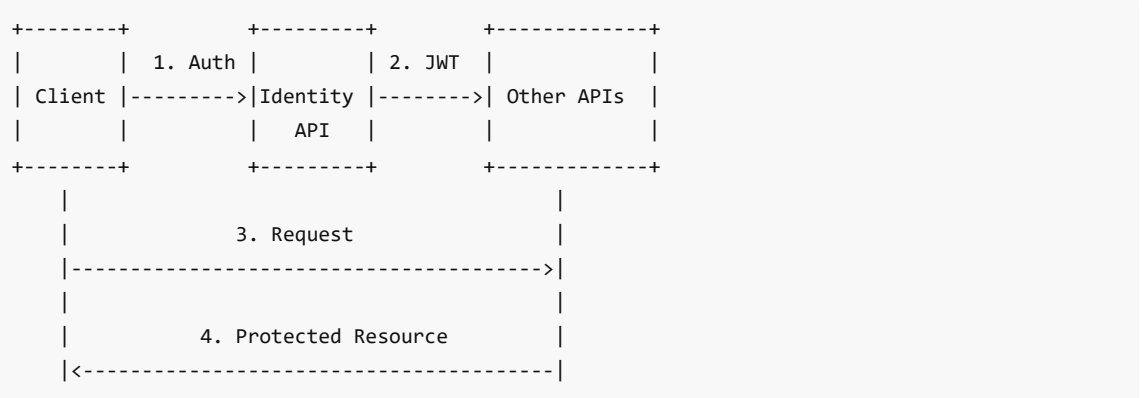
```
Tasks.Write.Api/
|
```

```
+-- Domain/
|   |-- Task.cs (Aggregate Root)
|   |-- Comment.cs (Entity)
|   `-- TaskStatus.cs (Value Object)
|
+-- Application/
|   |-- Commands/
|   |   |-- CreateTask/
|   |   |   |-- CreateTaskCommand.cs
|   |   |   `-- CreateTaskCommandHandler.cs
|   |   `-- UpdateStatus/
|   |       |-- UpdateTaskStatusCommand.cs
|   |       `-- UpdateTaskStatusCommandHandler.cs
|   `-- Events/
|       |-- TaskCreated.cs
|       `-- TaskStatusUpdated.cs
|
+-- Infrastructure/
|   |-- Persistence/
|   |   |-- TaskRepository.cs
|   |   `-- TaskContext.cs
|   `-- EventStore/
|       |-- RedisEventStore.cs
|       `-- EventPublisher.cs
|
`-- API/
    |-- Controllers/
    |   `-- TasksController.cs
    `-- DTOs/
        |-- CreateTaskRequest.cs
        `-- TaskResponse.cs
```

Data Flow Diagram



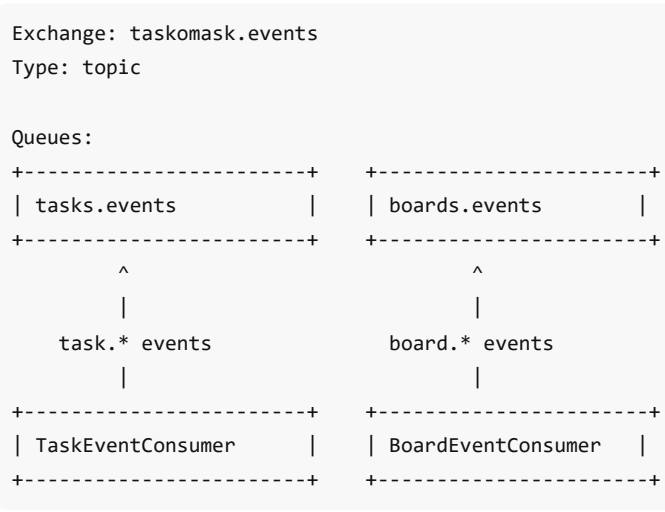
Authentication Flow



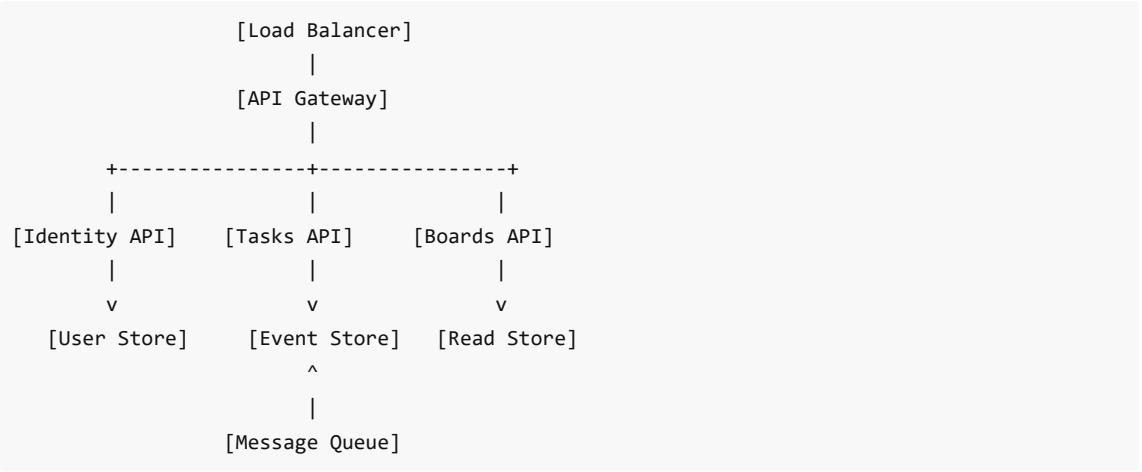
Event Store Structure (Redis)

```
Key Pattern: events:{entityType}:{entityId}
Example: events:task:123

[Newest Event] --> [Event N-1] --> ... --> [Event 1] --> [Event 0]

Event Structure:
{
    "id": "guid",
    "entityId": "123",
    "entityType": "task",
    "eventType": "TaskCreated",
    "data": { ... },
    "timestamp": "2025-01-24T18:47:10Z"
}
```

Message Queue Structure (RabbitMQ)



Deployment Architecture



Monitoring Setup