================================================================================
PROJECT DOCUMENTATION COVER PAGE
================================================================================

HIT SERVER-SIDE NODE PROJECT
Code Review & Documentation Package

================================================================================
DEVELOPMENT TEAM INFORMATION
================================================================================

TEAM MANAGER:
────────────────────────────────────────────────────────────────────────────────

Name:           Elad Hayek

================================================================================
TEAM MEMBERS:
────────────────────────────────────────────────────────────────────────────────

TEAM MEMBER 1:
Name:           Elad Hayek
ID:             211873542
Mobile Number:  0543106916
Email Address:  elad3283@gmail.com

TEAM MEMBER 2:
Name:           Ofir Zohar
ID:             318642980
Mobile Number:  0505913099
Email Address:  Ofirzohar11@gmail.com

TEAM MEMBER 3:
Name:           Zohar Abramoviz
ID:             209524156
Mobile Number:  0523805476
Email Address:  zohartalavoda@gmail.com

================================================================================
PROJECT VIDEO
================================================================================

https://youtu.be/C_Qln-5Neco

================================================================================
QUICK START GUIDE
================================================================================

Prerequisites:
  - Node.js (v14 or higher)
  - MongoDB (local instance or connection URI)

```
  - npm package manager

Installation:

  1. Navigate to project root:
     cd C:\<YOUR_LOCATION>\HIT-Server-side-node-project

  2. Install dependencies for each service:
     cd admin-service && npm install
     cd ../costs-service && npm install
     cd ../logs-service && npm install
     cd ../users-service && npm install

Running Services:

  Development Mode:
    Admin Service:    npm run dev (from admin-service directory, PORT 2000)
    Users Service:    npm run dev (from users-service directory, PORT 3000)
    Costs Service:    npm run dev (from costs-service directory, PORT 4000)
    Logs Service:     npm run dev (from logs-service directory, PORT 5000)

  Production Mode:
    npm start (from any service directory)

Testing:

  Run unit tests for a service:
    npm test (from service directory)

  Run tests in watch mode:
    npm run test:watch

Environment Variables:

  Each service can have an .env file with the following variables:
    NODE_ENV=development
    PORT=<service-specific-port>
    MONGO_URI=mongodb://localhost:27017/app
    LOGGING_SERVICE_URL=http://localhost:5000/api
    LOGGING_SERVICE_TIMEOUT=3000
    LOG_LEVEL=info

  Additional variables for costs-service:
    USERS_SERVICE_URL=http://localhost:3000/api
    USERS_SERVICE_TIMEOUT=3000

  Additional variables for users-service:
    COSTS_SERVICE_URL=http://localhost:4000/api
    COSTS_SERVICE_TIMEOUT=3000
```

```
================================================================================
COLLABORATIVE TOOLS SUMMARY & DEVELOPMENT PROCESS
================================================================================

GitHub was used as the main collaborative development tool, allowing team members
to work in parallel on separate features using branches,
manage version control, and safely merge changes through pull requests.
This enabled efficient coordination and prevented code conflicts.
Slack was used as the primary communication tool for the team,
supporting real-time discussions, task coordination, and quick problem solving.
Through Slack, we shared updates, divided responsibilities, and kept continuous
communication throughout the project,
which helped us work as a coordinated and organized team.

================================================================================
ADMIN SERVICE - COMPLETE CODE DOCUMENTATION
================================================================================

This file contains all source code files from the admin-service microservice,
organized by folder structure for easy navigation and code review.

================================================================================
FOLDER STRUCTURE
================================================================================

admin-service/
├── package.json
├── server.js
└── src/
    ├── app/
    │   ├── app.js
    │   ├── controllers/
    │   │   └── admin_controller.js
    │   ├── middlewares/
    │   │   ├── error_middleware.js
    │   │   └── logger_middleware.js
    │   ├── routes/
    │   │   ├── admin_routes.js
    │   │   └── index.js
    │   └── services/
    │       └── admin_service.js
    ├── clients/
    │   └── logging_client.js
    ├── config/
    │   └── index.js
    ├── db/
    │   └── models/
    │       └── index.js
    ├── errors/
    │   ├── app_error.js
```

```
    │   ├── duplicate_error.js
    │   ├── not_found_error.js
    │   ├── service_error.js
    │   └── validation_error.js
    └── logging/
        └── index.js
```

```
================================================================================
ADMIN SERVICE - COMPLETE CODE DOCUMENTATION
================================================================================

This file contains all source code files from the admin-service microservice,
organized by folder structure for easy navigation and code review.

================================================================================
FOLDER STRUCTURE
================================================================================

admin-service/
├── package.json
├── server.js
└── src/
    ├── app/
    │   ├── app.js
    │   ├── controllers/
    │   │   └── admin_controller.js
    │   ├── middlewares/
    │   │   ├── error_middleware.js
    │   │   └── logger_middleware.js
    │   ├── routes/
    │   │   ├── admin_routes.js
    │   │   └── index.js
    │   └── services/
    │       └── admin_service.js
    ├── clients/
    │   └── logging_client.js
    ├── config/
    │   └── index.js
    ├── db/
    │   └── models/
    │       └── index.js
    ├── errors/
    │   ├── app_error.js
    │   ├── duplicate_error.js
    │   ├── not_found_error.js
    │   ├── service_error.js
    │   └── validation_error.js
    └── logging/
        └── index.js
```

```
================================================================================
FILE: admin-service/package.json
================================================================================

{
  "name": "hit-users-service",
  "version": "1.0.0",
  "description": "Users microservice",
  "main": "server.js",
  "scripts": {
    "test": "cross-env NODE_ENV=test jest",
    "test:watch": "cross-env NODE_ENV=test jest --watch",
    "start": "node server.js",
    "dev": "cross-env PORT=2000 nodemon server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "commonjs",
  "dependencies": {
    "axios": "^1.7.9",
    "cross-env": "^7.0.3",
    "dotenv": "^17.2.3",
    "express": "^5.2.1",
    "mongoose": "^9.1.1",
    "pino": "^10.1.0",
    "pino-http": "^11.0.0"
  },
  "devDependencies": {
    "cross-env": "^10.1.0",
    "jest": "^30.2.0",
    "nodemon": "^3.1.11",
    "pino-pretty": "^13.1.3",
    "supertest": "^7.2.2"
  }
}

================================================================================
FILE: admin-service/server.js
================================================================================

// Server entry point - initializes and starts the Express application

// Load application factory from app configuration
const createApp = require("./src/app/app");
// Import logger instance for logging server events
const { logger } = require("./src/logging");
// Load environment configuration variables
const config = require("./src/config");
```

```
// Create Express application with all middleware and routes configured
const app = createApp();

// Extract port number from config or use default port 2000
const port = config.PORT || 2000;
// Start listening for HTTP requests on specified port
app.listen(port, () => {
  // Log server startup confirmation with port number
  logger.info(`Server running on port ${port}`);
});

================================================================================
FILE: admin-service/src/app/app.js
================================================================================

// Express application factory module - creates and configures the Express app
instance
const express = require("express");
const loggerMiddleware = require("./middlewares/logger_middleware");
const errorMiddleware = require("./middlewares/error_middleware");
const routes = require("./routes");

/**
 * Factory function that creates and configures Express application
 * Applies middleware stack and route handlers
 * @returns {Object} Configured Express application instance
 */
const createApp = function () {
  // Initialize Express application instance
  const app = express();
  // Apply HTTP request logging middleware first in the stack
  app.use(loggerMiddleware);
  // Parse incoming JSON request bodies to JavaScript objects
  app.use(express.json());
  // Mount API routes under /api path prefix
  app.use("/api", routes);
  // Apply error handling middleware at the end of the stack
  app.use(errorMiddleware);

  // Return configured application for server startup
  return app;
};

// Export app factory function for server initialization
module.exports = createApp;

================================================================================
FILE: admin-service/src/app/controllers/admin_controller.js
================================================================================
```

```
// Admin controller module - handles HTTP requests for admin endpoints
const adminService = require("../services/admin_service");

/**
 * Retrieves team members information endpoint
 * Handles GET /api/about requests
 * @param {Object} req - Express request object
 * @param {Object} res - Express response object containing JSON response
 * @param {Function} next - Express next middleware function for error handling
 * @returns {void} Sends JSON response with team members (200) or error to next
middleware
 */
const getAbout = function (req, res, next) {
  try {
    // Fetch hardcoded team members from service layer
    const teamMembers = adminService.getTeamMembers();
    // Return 200 success status with team members data
    res.status(200).json(teamMembers);
  } catch (error) {
    // Pass any errors to error handling middleware
    next(error);
  }
};

// Export controller functions for routing use
module.exports = {
  getAbout,
};

================================================================================
FILE: admin-service/src/app/services/admin_service.js
================================================================================

// Admin service module - contains business logic for admin operations

/**
 * Retrieves all team members information
 * Returns a hardcoded array of team members with their names
 * @returns {Array<Object>} Array of team member objects with first_name and
last_name properties
 */
const getTeamMembers = function () {
  // Initialize array of hardcoded team members
  const teamMembers = [
    {
      first_name: "Elad",
      last_name: "Hayek",
    },
    // Team member 2 - Ofir Zohar
    {
```

```
      first_name: "Ofir",
      last_name: "Zohar",
    },
    // Team member 3 - Zohar Abramoviz
    {
      first_name: "Zohar",
      last_name: "Abramoviz",
    },
  ];

  // Return the complete team members array to caller
  return teamMembers;
};

// Export service functions for use in controllers
module.exports = {
  getTeamMembers,
};

================================================================================
FILE: admin-service/src/app/routes/admin_routes.js
================================================================================

// Admin routes module - defines HTTP endpoints for admin service

// Import Express framework for route definition
const express = require("express");
// Import admin controller with route handler functions
const adminController = require("../controllers/admin_controller");

// Create Express router instance for admin routes
const router = express.Router();

// Define GET endpoint to retrieve team member information
// Route: GET /api/about
// Handler: Returns JSON array of team members with names
router.get("/about", adminController.getAbout);

// Export router for mounting in main app routes
module.exports = router;

================================================================================
FILE: admin-service/src/app/routes/index.js
================================================================================

// Main routes aggregation module - combines all API route modules

// Import Express router for route definition
const express = require("express");
// Import admin-specific routes module
```

```
const adminRoutes = require("./admin_routes");
// Create main router instance for API routes
const router = express.Router();

// Mount admin routes at root path (handles /about endpoint)
router.use("/", adminRoutes);

// Export aggregated routes for app mounting
module.exports = router;

================================================================================
FILE: admin-service/src/app/middlewares/error_middleware.js
================================================================================

// Error handling middleware module - processes and formats error responses

// Map error types to standardized error IDs for client identification
const ERROR_ID_MAP = {
  ValidationError: "ERR_VALIDATION_001",
  NotFoundError: "ERR_NOT_FOUND_002",
  DuplicateError: "ERR_DUPLICATE_003",
  ServiceError: "ERR_SERVICE_004",
  AppError: "ERR_APP_005",
};

/**
 * Maps error class name to standardized error ID
 * Used to identify error types in API responses
 * @param {Error} err - Error object with constructor name
 * @returns {string} Standardized error ID string
 */
const getErrorId = function (err) {
  // Extract error type name from error class constructor
  const errorType = err.constructor.name;
  // Return mapped ID or generic unknown error code
  return ERROR_ID_MAP[errorType] || "ERR_UNKNOWN_999";
};

/**
 * Express error handling middleware
 * Catches errors and sends formatted JSON response
 * @param {Error} err - Error object thrown during request processing
 * @param {Object} req - Express request object
 * @param {Object} res - Express response object for sending reply
 * @param {Function} next - Express next middleware function (unused in error
handler)
 * @returns {void} Sends JSON response with error details
 */
const errorHandler = function (err, req, res, next) {
  // Extract HTTP status code from error or default to 500
```

```
  const status = err.status || 500;
  // Get error message or provide default message
  const message = err.message || "Internal Server Error";
  // Map error type to standardized error ID
  const errorId = getErrorId(err);

  // Attach error to response object for logging middleware
  res.err = err;

  // Send error response in JSON format with status code
  res.status(status).json({
    id: errorId,
    message: message,
  });
};

// Export error handler middleware for Express app
module.exports = errorHandler;

================================================================================
FILE: admin-service/src/app/middlewares/logger_middleware.js
================================================================================

// HTTP request logging middleware module - logs all HTTP requests and responses

// Import pino HTTP logging plugin for request/response tracking
const pinoHttp = require("pino-http");
// Import logger instance configured for the application
const { logger } = require("../../logging");

// Configure pino HTTP logger with custom formatting and level determination
const httpLogger = pinoHttp({
  // Use application logger instance for output
  logger,
  /**
   * Determines appropriate log level based on HTTP response status
   * Client errors (4xx) log as warnings, server errors (5xx) as errors
   * @param {Object} req - Express request object
   * @param {Object} res - Express response object
   * @param {Error} err - Error object if present
   * @returns {string} Log level: "info", "warn", or "error"
   */
  customLogLevel: function (req, res, err) {
    // Log client errors (400-499) as warnings
    if (res.statusCode >= 400 && res.statusCode < 500) {
      return "warn";
    } else if (res.statusCode >= 500 || err || res.err) {
      // Log server errors (500+) and processing errors as errors
      return "error";
    }
```

```
    // Log successful requests (2xx, 3xx) as info
    return "info";
  },
  /**
   * Formats log message for successful requests
   * @param {Object} req - Express request object
   * @param {Object} res - Express response object with status code
   * @returns {string} Formatted log message with method, path, status
   */
  customSuccessMessage: function (req, res) {
    return `${req.method} ${req.url} ${res.statusCode}`;
  },
  /**
   * Formats log message for error responses
   * @param {Object} req - Express request object
   * @param {Object} res - Express response object
   * @param {Error} err - Error that occurred during request
   * @returns {string} Formatted error log with details
   */
  customErrorMessage: function (req, res, err) {
    // Extract error from parameter or response object
    const error = err || res.err;
    // Return formatted message with error details
    return `${req.method} ${req.url} ${res.statusCode} - ${error?.message}`;
  },
});

/**
 * Logging middleware function called for each HTTP request
 * Logs request/response information using pino
 * @param {Object} req - Express request object
 * @param {Object} res - Express response object
 * @param {Function} next - Express next middleware function
 * @returns {void} Passes control to next middleware
 */
const loggingMiddleware = function (req, res, next) {
  // Call pino HTTP logger to process request/response
  httpLogger(req, res, next);
};

// Export logging middleware for use in Express app
module.exports = loggingMiddleware;

================================================================================
FILE: admin-service/src/clients/logging_client.js
================================================================================

// Logging service client module - handles communication with external logging
service
```

```
// Import HTTP client for making requests to logging service
const axios = require("axios");
// Import configuration with logging service URL and timeout settings
const config = require("../config");

/**
 * Sends log data to external logging service via HTTP POST
 * Handles network errors gracefully without throwing
 * @param {Object} logData - Log object with level, message, timestamp, etc.
 * @returns {Promise<void>} Resolves after sending or silently fails
 */
const sendLogToService = async function (logData) {
  // Check if logging service URL is configured
  if (!config.LOGGING_SERVICE_URL) {
    // Exit early if service is not configured
    return;
  }

  try {
    // Send POST request to logging service endpoint
    await axios.post(`${config.LOGGING_SERVICE_URL}/logs`, logData, {
      // Set request timeout to prevent hanging
      timeout: config.LOGGING_SERVICE_TIMEOUT,
      // Specify JSON content type for the request
      headers: {
        "Content-Type": "application/json",
      },
    });
  } catch (error) {
    // Log error to console if service communication fails
    console.error("Failed to send log to logging service:", error.message);
  }
};

/**
 * Creates and sends a log entry to the logging service
 * Wraps log data and sends asynchronously
 * @param {Object} logData - Log object with level, message, and optional metadata
 * @returns {void} Sends log asynchronously without waiting
 */
const createLog = function (logData) {
  // Invoke async send function without awaiting to avoid blocking
  sendLogToService({
    // Spread existing log data
    ...logData,
    // Add current timestamp in ISO format
    timestamp: new Date().toISOString(),
  });
};
```

```
// Export logging client functions for use throughout application
module.exports = {
  createLog,
};

================================================================================
FILE: admin-service/src/config/index.js
================================================================================

// Environment configuration module - loads and exports application settings

// Load environment variables from .env file into process.env
require("dotenv").config();

// Export configuration object with environment-specific settings
const env = {
  // Application environment (development, production, test)
  NODE_ENV: process.env.NODE_ENV || "development",
  // Server port for HTTP listener
  PORT: process.env.PORT || 2000,
  // Log level for Pino logger (debug, info, warn, error)
  LOG_LEVEL: process.env.LOG_LEVEL || "info",
  // MongoDB connection URI for database access
  MONGO_URI: process.env.MONGO_URI || "mongodb://localhost:27017/app",
  // External logging service base URL for log submission
  LOGGING_SERVICE_URL:
    process.env.LOGGING_SERVICE_URL || "http://localhost:5000/api",
  // Request timeout for logging service communication in milliseconds
  LOGGING_SERVICE_TIMEOUT: process.env.LOGGING_SERVICE_TIMEOUT || 3000,
};

// Export configuration for use throughout application
module.exports = env;

================================================================================
FILE: admin-service/src/db/models/index.js
================================================================================

// Database models aggregation module - exports all Mongoose models

// Note: No models are used in admin service
// Team members data is hardcoded in the service layer
// This file is maintained for consistency with architecture

// Export empty models object as all data is in memory
module.exports = {};

================================================================================
FILE: admin-service/src/errors/app_error.js
================================================================================
```

```
// Application error class module - defines base error for application

/**
 * Base error class for all application errors
 * Extends native JavaScript Error to add HTTP status codes
 * All custom errors should inherit from this class
 */
class AppError extends Error {
  /**
   * Constructor for creating application error instances
   * @param {string} message - Human-readable error description
   * @param {number} status - HTTP status code (default 500)
   */
  constructor(message, status) {
    // Call parent Error constructor with message
    super(message);
    // Store HTTP status code for response handling
    this.status = status;
  }
}

// Export AppError class for inheritance by specific error types
module.exports = { AppError };

================================================================================
FILE: admin-service/src/errors/duplicate_error.js
================================================================================

// Duplicate error module - handles resource conflict scenarios (HTTP 409)
// Used when attempting to create a resource that already exists (duplicate ID,
email, etc.)

// Import base AppError class for inheritance
const { AppError } = require("./app_error");

/**
 * DuplicateError class for resource conflict/duplication scenarios
 * Extends AppError with HTTP 409 (Conflict) status code
 * Used when client attempts to create a resource with duplicate unique fields
 */
class DuplicateError extends AppError {
  /**
   * Constructor for creating duplicate error instances
   * @param {string} message - Human-readable error description of the duplication
   */
  constructor(message) {
    // Call parent with message and HTTP 409 Conflict status
    super(message, 409);
    // Set error name for debugging and error identification
```

```
    this.name = "DuplicateError";
  }
}

// Export DuplicateError class for use in services and controllers
module.exports = { DuplicateError };

================================================================================
FILE: admin-service/src/errors/not_found_error.js
================================================================================

// Not found error module - handles resource not found scenarios (HTTP 404)
// Used when requested resource (user, cost, log) does not exist in database

// Import base AppError class for inheritance
const { AppError } = require("./app_error");

/**
 * NotFoundError class for missing resource scenarios
 * Extends AppError with HTTP 404 (Not Found) status code
 * Used when client requests a resource that doesn't exist
 */
class NotFoundError extends AppError {
  /**
   * Constructor for creating not found error instances
   * @param {string} message - Human-readable error description of what was not
found
   */
  constructor(message) {
    // Call parent with message and HTTP 404 Not Found status
    super(message, 404);
    // Set error name for debugging and error identification
    this.name = "NotFoundError";
  }
}

// Export NotFoundError class for use in services and controllers
module.exports = { NotFoundError };

================================================================================
FILE: admin-service/src/errors/service_error.js
================================================================================

// Service error module - handles inter-service communication failures (HTTP
502/503)
// Used when external services (users, costs, logging) are unavailable or fail

// Import base AppError class for inheritance
const { AppError } = require("./app_error");
```

```
/**
 * ServiceError class for inter-service communication failures
 * Extends AppError with configurable HTTP status code (default 502 Bad Gateway)
 * Used when external services are unreachable, timeout, or return errors
 */
class ServiceError extends AppError {
  /**
   * Constructor for creating service error instances
   * @param {string} message - Human-readable error description of service failure
   * @param {number} status - HTTP status code (502 Bad Gateway or 503 Service
Unavailable), default 502
   */
  constructor(message, status = 502) {
    // Call parent with message and provided or default HTTP 502 status
    super(message, status);
    // Set error name for debugging and error identification
    this.name = "ServiceError";
  }
}

// Export ServiceError class for use in services and controllers
module.exports = { ServiceError };

================================================================================
FILE: admin-service/src/errors/validation_error.js
================================================================================

// Validation error module - handles request validation failures (HTTP 400)
// Used when input data fails schema validation or constraint checks

// Import base AppError class for inheritance
const { AppError } = require("./app_error");

/**
 * ValidationError class for data validation failures
 * Extends AppError with HTTP 400 (Bad Request) status code
 * Used when client sends invalid data that fails validation rules
 */
class ValidationError extends AppError {
  /**
   * Constructor for creating validation error instances
   * @param {string} message - Human-readable error description of validation
failure
   */
  constructor(message) {
    // Call parent with message and HTTP 400 Bad Request status
    super(message, 400);
    // Set error name for debugging and error identification
    this.name = "ValidationError";
  }
```

```
}

// Export ValidationError class for use in services and controllers
module.exports = { ValidationError };

================================================================================
FILE: admin-service/src/logging/index.js
================================================================================

// Logger configuration module - sets up Pino logging with custom streams

// Import Pino logging library for structured logging
const pino = require("pino");
// Import logging client for sending logs to external service
const loggingClient = require("../clients/logging_client");
// Import configuration for LOG_LEVEL setting
const config = require("../config");

// Map Pino numeric log levels to string representations for storage
const pinoLevelToString = {
  10: "debug", // trace level (Pino 10)
  20: "debug", // debug level (Pino 20)
  30: "info", // info level (Pino 30)
  40: "warn", // warn level (Pino 40)
  50: "error", // error level (Pino 50)
  60: "error", // fatal level (Pino 60)
};

/**
 * Custom stream object for writing logs to external service
 * Implements write(msg) interface for Pino streams
 */
const customStream = {
  /**
   * Writes log message to logging service via HTTP
   * Parses Pino JSON output and sends to service
   * @param {string} msg - JSON formatted log message from Pino
   * @returns {void}
   */
  write: (msg) => {
    try {
      // Parse Pino log message JSON string to object
      const logObj = JSON.parse(msg);
      // Convert numeric Pino level to string level
      const level = pinoLevelToString[logObj.level] || "info";

      // Send structured log to logging service endpoint
      loggingClient.createLog({
        // Log severity level (info, warn, error, debug)
        level,
```

```
        // Log message text content
        message: logObj.msg,
        // Log timestamp in Date format
        timestamp: new Date(logObj.time),
        // HTTP method from request (if available)
        method: logObj.req?.method,
        // URL path from request (if available)
        url: logObj.req?.url,
        // HTTP response status code (if available)
        statusCode: logObj.res?.statusCode,
        // Request processing time in milliseconds
        responseTime: logObj.responseTime,
      });
    } catch (err) {
      // Fallback error logging if processing fails
      console.error("Failed to send log:", err.message);
    }
  },
};

// Create Pino logger with console and custom stream outputs
const logger = pino(
  // Configure base logger with configurable log level
  { level: config.LOG_LEVEL },
  // Use multiple output streams simultaneously
  pino.multistream([
    // Stream 1: Output to standard output (console)
    { level: config.LOG_LEVEL, stream: process.stdout },
    // Stream 2: Output to custom logging service client
    { level: config.LOG_LEVEL, stream: customStream },
  ])
);

// Export logger instance for use throughout application
module.exports = { logger };

================================================================================
FILE: admin-service/tests/unit/admin.test.js
================================================================================

// Unit test suite for admin service - tests team members retrieval

// Mock logging client to prevent external HTTP calls during testing
jest.mock("../../src/clients/logging_client", () => ({
  logRequest: jest.fn(),
  logResponse: jest.fn(),
  logCustom: jest.fn(),
}));

// Import the admin service module under test
```

```
const adminService = require("../../src/app/services/admin_service");

/**
 * Test suite for admin service functionality
 * Tests the getTeamMembers function behavior
 */
describe("Admin Service", () => {
  /**
   * Test suite for getTeamMembers function
   */
  describe("getTeamMembers", () => {
    /**
     * Test 1: Verify function returns an array with team members
     */
    test("should return hardcoded team members", () => {
      // Call getTeamMembers function
      const result = adminService.getTeamMembers();

      // Verify result is an array instance
      expect(result).toBeInstanceOf(Array);
      // Verify array contains at least one team member
      expect(result.length).toBeGreaterThan(0);
      // Verify first member has first_name property
      expect(result[0]).toHaveProperty("first_name");
      // Verify first member has last_name property
      expect(result[0]).toHaveProperty("last_name");
    });

    /**
     * Test 2: Validate team member object structure
     * Each member should have first_name and last_name strings
     */
    test("should return team members with correct structure", () => {
      // Get team members from service
      const result = adminService.getTeamMembers();

      // Iterate through each team member
      result.forEach((member) => {
        // Verify member has exactly two properties with string values
        expect(member).toEqual({
          first_name: expect.any(String),
          last_name: expect.any(String),
        });
      });
    });

    /**
     * Test 3: Ensure function returns consistent data
     * Multiple calls should return identical arrays
     */
```

```
    test("should return consistent team members on multiple calls", () => {
      // Call getTeamMembers first time
      const result1 = adminService.getTeamMembers();
      // Call getTeamMembers second time
      const result2 = adminService.getTeamMembers();

      // Verify both calls return identical data
      expect(result1).toEqual(result2);
    });
  });
});

================================================================================
FILE: admin-service/tests/unit/admin.test.js
================================================================================

// Unit test suite for admin service - tests team members retrieval

// Mock logging client to prevent external HTTP calls during testing
jest.mock("../../src/clients/logging_client", () => ({
  logRequest: jest.fn(),
  logResponse: jest.fn(),
  logCustom: jest.fn(),
}));

// Import the admin service module under test
const adminService = require("../../src/app/services/admin_service");

/**
 * Test suite for admin service functionality
 * Tests the getTeamMembers function behavior
 */
describe("Admin Service", () => {
  /**
   * Test suite for getTeamMembers function
   */
  describe("getTeamMembers", () => {
    /**
     * Test 1: Verify function returns an array with team members
     */
    test("should return hardcoded team members", () => {
      // Call getTeamMembers function
      const result = adminService.getTeamMembers();

      // Verify result is an array instance
      expect(result).toBeInstanceOf(Array);
      // Verify array contains at least one team member
      expect(result.length).toBeGreaterThan(0);
      // Verify first member has first_name property
      expect(result[0]).toHaveProperty("first_name");
```

```
      // Verify first member has last_name property
      expect(result[0]).toHaveProperty("last_name");
    });

    /**
     * Test 2: Validate team member object structure
     * Each member should have first_name and last_name strings
     */
    test("should return team members with correct structure", () => {
      // Get team members from service
      const result = adminService.getTeamMembers();

      // Iterate through each team member
      result.forEach((member) => {
        // Verify member has exactly two properties with string values
        expect(member).toEqual({
          first_name: expect.any(String),
          last_name: expect.any(String),
        });
      });
    });

    /**
     * Test 3: Ensure function returns consistent data
     * Multiple calls should return identical arrays
     */
    test("should return consistent team members on multiple calls", () => {
      // Call getTeamMembers first time
      const result1 = adminService.getTeamMembers();
      // Call getTeamMembers second time
      const result2 = adminService.getTeamMembers();

      // Verify both calls return identical data
      expect(result1).toEqual(result2);
    });
  });
});

================================================================================
END OF ADMIN SERVICE CODE DOCUMENTATION
================================================================================

================================================================================
COSTS SERVICE - COMPLETE CODE DOCUMENTATION
================================================================================

This file contains all source code files from the costs-service microservice,
organized by folder structure for easy navigation and code review.

================================================================================
```

FOLDER STRUCTURE
================================================================================

```
costs-service/
├── package.json
├── server.js
└── src/
    ├── app/
    │   ├── app.js
    │   ├── controllers/
    │   │   └── costs_controller.js
    │   ├── middlewares/
    │   │   ├── error_middleware.js
    │   │   └── logger_middleware.js
    │   ├── repositories/
    │   │   └── costs_repository.js
    │   ├── routes/
    │   │   ├── costs_routes.js
    │   │   └── index.js
    │   └── services/
    │       └── costs_service.js
    ├── clients/
    │   ├── logging_client.js
    │   └── users_client.js
    ├── config/
    │   ├── cost_categories.js
    │   └── index.js
    ├── db/
    │   ├── index.js
    │   └── models/
    │       ├── cost.model.js
    │       ├── monthlyReport.model.js
    │       └── index.js
    ├── errors/
    │   ├── app_error.js
    │   ├── duplicate_error.js
    │   ├── not_found_error.js
    │   ├── service_error.js
    │   └── validation_error.js
    └── logging/
        └── index.js
```

================================================================================
FILE: costs-service/package.json
================================================================================

```
{
  "name": "hit-costs-service",
  "version": "1.0.0",
  "description": "Costs microservice",
```

```
  "main": "server.js",
  "scripts": {
    "test": "cross-env NODE_ENV=test jest",
    "test:watch": "cross-env NODE_ENV=test jest --watch",
    "start": "node server.js",
    "dev": "cross-env PORT=4000 nodemon server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "commonjs",
  "dependencies": {
    "axios": "^1.7.9",
    "cross-env": "^7.0.3",
    "dotenv": "^17.2.3",
    "express": "^5.2.1",
    "mongoose": "^9.1.1",
    "pino": "^10.1.0",
    "pino-http": "^11.0.0"
  },
  "devDependencies": {
    "cross-env": "^10.1.0",
    "jest": "^30.2.0",
    "nodemon": "^3.1.11",
    "pino-pretty": "^13.1.3",
    "supertest": "^7.2.2"
  }
}

================================================================================
FILE: costs-service/server.js
================================================================================

// Server entry point for costs service - initializes database and starts
application

// Load Express application factory
const createApp = require("./src/app/app");
// Import database connection function for MongoDB initialization
const { connectDB } = require("./src/db");
// Import logger instance for logging server events
const { logger } = require("./src/logging");
// Load environment configuration and settings
const config = require("./src/config");

// Create Express application with middleware and routes
const app = createApp();

// Connect to MongoDB database before starting server
connectDB().then(() => {
```

```
  // Extract port from config or use default 4000
  const port = config.PORT || 4000;
  // Start HTTP server listening on configured port
  app.listen(port, () => {
    // Log confirmation of successful server startup
    logger.info(`Server running on port ${port}`);
  });
});

================================================================================
FILE: costs-service/src/app/app.js
================================================================================

// Express application factory module - creates and configures costs service app

// Import Express framework for application creation
const express = require("express");
// Import HTTP request logging middleware
const loggerMiddleware = require("./middlewares/logger_middleware");
// Import error handling middleware for response
const errorMiddleware = require("./middlewares/error_middleware");
// Import all API routes configuration
const routes = require("./routes");

/**
 * Factory function that creates and configures Express application
 * Sets up middleware stack and mounts API routes
 * @returns {Object} Configured Express application instance
 */
const createApp = function () {
  // Create new Express application instance
  const app = express();
  // Apply request logging middleware at the beginning
  app.use(loggerMiddleware);
  // Parse incoming JSON request bodies
  app.use(express.json());
  // Mount API routes under /api path
  app.use("/api", routes);
  // Apply error handling middleware at the end
  app.use(errorMiddleware);

  // Return fully configured application
  return app;
};

// Export app factory function for server startup
module.exports = createApp;

================================================================================
FILE: costs-service/src/app/controllers/costs_controller.js
```

```
================================================================================

// Costs controller module - handles HTTP requests for cost management endpoints

// Import costs service for business logic access
const costsService = require("../services/costs_service");

/**
 * Creates a new cost entry in the database
 * Handles POST /api/add requests with cost data
 * @param {Object} req - Express request object with cost data in body
 * @param {Object} res - Express response object for sending response
 * @param {Function} next - Express next middleware function for error handling
 * @returns {void} Sends 201 created status with cost data or error to next
 */
const addCost = async function (req, res, next) {
  try {
    // Validate and create cost through service layer
    const cost = await costsService.createCost(req.body);
    // Return 201 Created status with the newly created cost object
    res.status(201).json(cost);
  } catch (err) {
    // Pass validation or service errors to error middleware
    next(err);
  }
};

/**
 * Retrieves monthly cost summary report for a user
 * Handles GET /api/report requests with query parameters
 * @param {Object} req - Express request object with query params (id, year, month)
 * @param {Object} res - Express response object for sending response
 * @param {Function} next - Express next middleware function for error handling
 * @returns {void} Sends 200 OK with report data or error to next
 */
const getMonthlyReport = async function (req, res, next) {
  try {
    // Fetch monthly cost report from service layer
    const report = await costsService.getMonthlyReport(req.query);
    // Return 200 OK status with monthly report data
    res.status(200).json(report);
  } catch (err) {
    // Pass validation or service errors to error middleware
    next(err);
  }
};

/**
 * Calculates total costs for a specific user
 * Handles GET /api/user-total requests
```

```
 * @param {Object} req - Express request object with userId query param
 * @param {Object} res - Express response object for sending response
 * @param {Function} next - Express next middleware function for error handling
 * @returns {void} Sends 200 OK with total costs or error to next
 */
const getUserTotalCosts = async function (req, res, next) {
  try {
    // Calculate user's total costs from service layer
    const totalCosts = await costsService.getUserTotalCosts(req.query);
    // Return 200 OK status with total costs data
    res.status(200).json(totalCosts);
  } catch (err) {
    // Pass validation or service errors to error middleware
    next(err);
  }
};

// Export controller functions for route handlers
module.exports = {
  addCost,
  getMonthlyReport,
  getUserTotalCosts,
};

================================================================================
FILE: costs-service/src/app/services/costs_service.js (Part 1 of 2)
================================================================================

// Costs service - business logic layer
// Handles cost validation, processing, and monthly report generation with user
verification
const costsRepository = require("../repositories/costs_repository");
const usersClient = require("../../clients/users_client");
const { logger } = require("../../logging");
const { ValidationError } = require("../../errors/validation_error");
const { NotFoundError } = require("../../errors/not_found_error");
const { ServiceError } = require("../../errors/service_error");
const Cost = require("../../db/models/cost.model");

/**
 * Validates cost data against schema and verifies user exists
 * Checks Cost model constraints and communicates with users service
 * @param {Object} data - Cost data object with userid, category, sum, description
 * @returns {Promise<Object>} Validated cost object if all checks pass
 * @throws {ValidationError} If cost data fails schema validation or user doesn't
exist
 * @throws {ServiceError} If users service is unavailable or returns error
 */
const validateCostData = async function (data) {
  // Create temporary cost to validate against Mongoose schema
```

```
  const tempCost = new Cost(data);
  const validationError = tempCost.validateSync();

  // If schema validation fails, extract and throw the first error
  if (validationError) {
    const firstErrorKey = Object.keys(validationError.errors)[0];
    const firstError = validationError.errors[firstErrorKey];
    throw new ValidationError(firstError.message);
  }

  // Check if the userid exists in the users service
  try {
    const userExists = await usersClient.checkUserExists(data.userid);
    if (!userExists) {
      throw new ValidationError(`User with id ${data.userid} does not exist`);
    }
  } catch (error) {
    // Re-throw validation errors as-is
    if (error instanceof ValidationError) {
      throw error;
    }
    // Handle service communication errors (connection refused, timeout, 5xx
errors)
    if (
      error.code === "ECONNREFUSED" ||
      error.code === "ETIMEDOUT" ||
      error.response?.status >= 500
    ) {
      logger.error(
        { userId: data.userid, error: error.message },
        "Users service unavailable"
      );
      throw new ServiceError("Users service unavailable", 503);
    } else if (error.response) {
      // Handle HTTP error responses from users service
      logger.error(
        { userId: data.userid, error: error.message },
        "Users service error"
      );
      throw new ServiceError(
        `Users service error: ${error.response.statusText}`,
        502
      );
    } else {
      // Handle unexpected errors during user verification
      logger.error(
        { userId: data.userid, error: error.message },
        "Failed to verify user existence"
      );
      throw new ServiceError("Failed to verify user existence", 502);
```

```
    }
  }

  return tempCost;
};

/**
 * Validates and normalizes parameters for monthly report endpoint
 * Ensures id, year, and month are valid numbers within acceptable ranges
 * @param {Object} params - Request parameters with id, year, month fields
 * @returns {Object} Validated and normalized object with userid, year, month as
numbers
 * @throws {ValidationError} If parameters are missing or invalid
 */
const validateMonthlyReportParams = function (params) {
  const { id, year, month } = params;

  // Validate user ID parameter
  if (id === undefined || id === null) {
    throw new ValidationError("Field 'id' is required and must be a number");
  }

  const userIdNum = Number(id);
  if (isNaN(userIdNum)) {
    throw new ValidationError("Field 'id' is required and must be a number");
  }

  // Validate year parameter (reasonable range: 1900-2100)
  const yearNum = Number(year);
  if (
    year === undefined ||
    year === null ||
    isNaN(yearNum) ||
    yearNum < 1900 ||
    yearNum > 2100
  ) {
    throw new ValidationError(
      "Field 'year' is required and must be a valid year"
    );
  }

  // Validate month parameter (must be 1-12)
  const monthNum = Number(month);
  if (
    month === undefined ||
    month === null ||
    isNaN(monthNum) ||
    monthNum < 1 ||
    monthNum > 12
  ) {
```

```
    throw new ValidationError(
      "Field 'month' is required and must be between 1 and 12"
    );
  }

  return {
    userid: userIdNum,
    year: yearNum,
    month: monthNum,
  };
};

/**
 * Validates and normalizes parameters for user total costs endpoint
 * Ensures userId is a valid number
 * @param {Object} params - Request parameters with userId field
 * @returns {number} Validated userId as a number
 * @throws {ValidationError} If userId is missing or not a valid number
 */
const validateUserTotalCostsParams = function (params) {
  const { userId } = params;

  // Validate userId parameter presence and type
  if (userId === undefined || userId === null) {
    throw new ValidationError(
      "Field 'userId' is required and must be a number"
    );
  }

  const userIdNum = Number(userId);
  if (isNaN(userIdNum)) {
    throw new ValidationError(
      "Field 'userId' is required and must be a number"
    );
  }

  return userIdNum;
};

/**
 * Creates a new cost entry with validation and returns formatted response
 * Validates cost data and user existence before persisting to database
 * @param {Object} costData - Cost object with userid, category, sum, description
 * @returns {Promise<Object>} Formatted cost response with _id, description,
category, userid, sum, createdAt
 * @throws {ValidationError} If cost data is invalid or user doesn't exist
 * @throws {ServiceError} If users service fails or database operation fails
 */
const createCost = async function (costData) {
  // Validate cost data against schema and verify user exists
```

```
  const validatedCost = await validateCostData(costData);

  // Persist cost to database with validated data
  const cost = await costsRepository.createCost({
    description: validatedCost.description,
    category: validatedCost.category,
    userid: validatedCost.userid,
    sum: validatedCost.sum,
  });

  // Return formatted response with all relevant fields
  return {
    _id: cost._id,
    description: cost.description,
    category: cost.category,
    userid: cost.userid,
    sum: cost.sum,
    createdAt: cost.createdAt,
  };
};

/**
 * Generates monthly cost report with caching for past months
 * Returns empty report for future dates, fresh data for current month, cached data
for past months
 * @param {Object} params - Request parameters with id, year, month
 * @returns {Promise<Object>} Object with userid, year, month, and costs array by
category
 * @throws {NotFoundError} If user doesn't exist
 * @throws {ValidationError} If parameters are invalid
 * @throws {ServiceError} If users service fails
 */
const getMonthlyReport = async function (params) {
  // Validate and normalize all parameters
  const { userid: id, year, month } = validateMonthlyReportParams(params);

  // Additional service implementation continues...
};

/**
 * Calculates total costs for a user
 * @param {Object} params - Request parameters with userId field
 * @returns {Promise<Object>} Object with total_costs
 * @throws {ValidationError} If userId is invalid
 */
const getUserTotalCosts = async function (params) {
  const userId = validateUserTotalCostsParams(params);
  const total = await costsRepository.getCostsTotalByUserId(userId);
  return { total_costs: total };
};
```

```
module.exports = {
  createCost,
  getMonthlyReport,
  getUserTotalCosts,
};

================================================================================
FILE: costs-service/src/app/repositories/costs_repository.js
================================================================================

// Costs repository - database access layer for cost management
// Handles all database operations for costs and monthly reports
const Cost = require("../../db/models/cost.model");
const MonthlyReport = require("../../db/models/monthlyReport.model");

/**
 * Creates a new cost entry in the database
 * @param {Object} costData - Cost object containing sum, category, description,
userid, createdAt
 * @returns {Promise<Object>} The saved cost document with _id and timestamps
 */
const createCost = async function (costData) {
  const cost = new Cost(costData);
  return await cost.save();
};

/**
 * Retrieves all costs associated with a specific user
 * @param {number} userid - The user ID to filter costs by
 * @returns {Promise<Array>} Array of cost documents for the user
 */
const findCostsByUserId = async function (userid) {
  return await Cost.find({ userid });
};

/**
 * Calculates total sum of all costs for a specific user
 * Uses MongoDB aggregation pipeline for efficient calculation
 * @param {number} userid - The user ID to calculate totals for
 * @returns {Promise<number>} Total sum of all user costs, returns 0 if no costs
exist
 */
const getCostsTotalByUserId = async function (userid) {
  // Aggregate pipeline: match user costs and group to sum them
  const result = await Cost.aggregate([
    {
      $match: {
        userid: Number(userid),
      },
```

```
    },
    {
      $group: {
        _id: null,
        total: { $sum: "$sum" },
      },
    },
  ]);

  // Return total or 0 if no results found
  return result.length > 0 ? result[0].total : 0;
};

/**
 * Retrieves cached monthly report for a user from the database
 * @param {number} userid - The user ID
 * @param {number} year - The report year (e.g., 2024)
 * @param {number} month - The report month (1-12)
 * @returns {Promise<Object|null>} Monthly report document or null if not found
 */
const getMonthlyReportFromCache = async function (userid, year, month) {
  return await MonthlyReport.findOne({
    userid: Number(userid),
    year: Number(year),
    month: Number(month),
  });
};

/**
 * Fetches and aggregates costs by category for a specific month
 * Groups costs by category and extracts day information from timestamps
 * @param {number} userid - The user ID to get costs for
 * @param {number} year - The year to filter by
 * @param {number} month - The month to filter by (1-12)
 * @returns {Promise<Array>} Array containing single object with category arrays
 */
const getCostsByMonthAggregation = async function (userid, year, month) {
  // Create date range boundaries for the specified month
  const startDate = new Date(year, month - 1, 1);
  const endDate = new Date(year, month, 1);

  // Query costs within the date range for the user
  const costs = await Cost.find({
    userid: Number(userid),
    createdAt: {
      $gte: startDate,
      $lt: endDate,
    },
  }).lean();
```

```
  // Initialize the costs structure with all categories
  const costsByCategory = {
    food: [],
    education: [],
    health: [],
    housing: [],
    sports: [],
  };

  // Group costs by category and extract day from createdAt
  costs.forEach((cost) => {
    const day = new Date(cost.createdAt).getDate();
    const costItem = {
      sum: cost.sum,
      description: cost.description,
      day,
    };
    costsByCategory[cost.category].push(costItem);
  });

  // Return in the expected format: array of objects with category keys
  return [
    {
      food: costsByCategory.food,
      education: costsByCategory.education,
      health: costsByCategory.health,
      housing: costsByCategory.housing,
      sports: costsByCategory.sports,
    },
  ];
};

/**
 * Caches a monthly cost report in the database
 * Stores pre-calculated monthly report data for quick retrieval
 * @param {number} userid - The user ID to cache report for
 * @param {number} year - The year of the report
 * @param {number} month - The month of the report (1-12)
 * @param {Object} data - The pre-calculated cost data by category
 * @returns {Promise<Object>} The created monthly report document
 */
const cacheMonthlyReport = async function (userid, year, month, data) {
  return await MonthlyReport.create({
    userid: Number(userid),
    year: Number(year),
    month: Number(month),
    costs: data,
  });
};
```

```
module.exports = {
  createCost,
  findCostsByUserId,
  getCostsTotalByUserId,
  getMonthlyReportFromCache,
  getCostsByMonthAggregation,
  cacheMonthlyReport,
};

================================================================================
FILE: costs-service/src/app/routes/costs_routes.js
================================================================================

// Costs routes module - defines HTTP endpoints for cost management
// Provides routes for adding costs, retrieving reports, and calculating totals

// Import Express framework for route definition
const express = require("express");
// Import costs controller with route handler functions
const costsController = require("../controllers/costs_controller");

// Create Express router instance for costs routes
const router = express.Router();

// POST endpoint to add a new cost entry
// Route: POST /api/add
// Handler: Validates cost data, checks user exists, and persists to database
router.post("/add", costsController.addCost);

// GET endpoint to retrieve monthly cost report by category
// Route: GET /api/report?id=<userid>&year=<year>&month=<month>
// Handler: Returns aggregated costs organized by category for specified month
router.get("/report", costsController.getMonthlyReport);

// GET endpoint to get total costs for a user
// Route: GET /api/user-total?userId=<userid>
// Handler: Calculates and returns sum of all costs for user
router.get("/user-total", costsController.getUserTotalCosts);

// Export router for mounting in main routes aggregation
module.exports = router;

================================================================================
FILE: costs-service/src/app/routes/index.js
================================================================================

// Routes aggregation module - combines all API route modules for costs service
// Provides centralized routing configuration for all cost-related endpoints

// Import Express framework for route definition
```

```
const express = require("express");
// Import costs-specific routes module containing all cost endpoint handlers
const costsRoutes = require("./costs_routes");
// Create main router instance for API routes
const router = express.Router();

// Mount costs routes at root path (all cost endpoints available at /api/...)
router.use("/", costsRoutes);

// Export aggregated routes for app mounting
module.exports = router;

================================================================================
FILE: costs-service/src/app/middlewares/error_middleware.js
================================================================================

// Error handling middleware module - processes and formats error responses

// Map error types to standardized error IDs for client identification
const ERROR_ID_MAP = {
  ValidationError: "ERR_VALIDATION_001",
  NotFoundError: "ERR_NOT_FOUND_002",
  DuplicateError: "ERR_DUPLICATE_003",
  ServiceError: "ERR_SERVICE_004",
  AppError: "ERR_APP_005",
};

/**
 * Maps error class name to standardized error ID
 * Used to identify error types in API responses
 * @param {Error} err - Error object with constructor name
 * @returns {string} Standardized error ID string
 */
const getErrorId = function (err) {
  // Extract error type name from error class constructor
  const errorType = err.constructor.name;
  // Return mapped ID or generic unknown error code
  return ERROR_ID_MAP[errorType] || "ERR_UNKNOWN_999";
};

/**
 * Express error handling middleware
 * Catches errors and sends formatted JSON response
 * @param {Error} err - Error object thrown during request processing
 * @param {Object} req - Express request object
 * @param {Object} res - Express response object for sending reply
 * @param {Function} next - Express next middleware function (unused in error
handler)
 * @returns {void} Sends JSON response with error details
 */
```

```
const errorHandler = function (err, req, res, next) {
  // Extract HTTP status code from error or default to 500
  const status = err.status || 500;
  // Get error message or provide default message
  const message = err.message || "Internal Server Error";
  // Map error type to standardized error ID
  const errorId = getErrorId(err);

  // Attach error to response object for logging middleware
  res.err = err;

  // Send error response in JSON format with status code
  res.status(status).json({
    id: errorId,
    message: message,
  });
};

module.exports = errorHandler;

================================================================================
FILE: costs-service/src/app/middlewares/logger_middleware.js
================================================================================

// HTTP request logging middleware module - logs all HTTP requests and responses

// Import pino HTTP logging plugin for request/response tracking
const pinoHttp = require("pino-http");
// Import logger instance configured for the application
const { logger } = require("../../logging");

// Configure pino HTTP logger with custom formatting and level determination
const httpLogger = pinoHttp({
  // Use application logger instance for output
  logger,
  /**
   * Determines appropriate log level based on HTTP response status
   * Client errors (4xx) log as warnings, server errors (5xx) as errors
   * @param {Object} req - Express request object
   * @param {Object} res - Express response object
   * @param {Error} err - Error object if present
   * @returns {string} Log level: "info", "warn", or "error"
   */
  customLogLevel: function (req, res, err) {
    // Log client errors (400-499) as warnings
    if (res.statusCode >= 400 && res.statusCode < 500) {
      return "warn";
    } else if (res.statusCode >= 500 || err || res.err) {
      // Log server errors (500+) and processing errors as errors
      return "error";
```

```
    }
    // Log successful requests (2xx, 3xx) as info
    return "info";
  },
  /**
   * Formats log message for successful requests
   * @param {Object} req - Express request object
   * @param {Object} res - Express response object with status code
   * @returns {string} Formatted log message with method, path, status
   */
  customSuccessMessage: function (req, res) {
    return `${req.method} ${req.url} ${res.statusCode}`;
  },
  /**
   * Formats log message for error responses
   * @param {Object} req - Express request object
   * @param {Object} res - Express response object
   * @param {Error} err - Error that occurred during request
   * @returns {string} Formatted error log with details
   */
  customErrorMessage: function (req, res, err) {
    // Extract error from parameter or response object
    const error = err || res.err;
    // Return formatted message with error details
    return `${req.method} ${req.url} ${res.statusCode} - ${error?.message}`;
  },
});

/**
 * Logging middleware function called for each HTTP request
 * Logs request/response information using pino
 * @param {Object} req - Express request object
 * @param {Object} res - Express response object
 * @param {Function} next - Express next middleware function
 * @returns {void} Passes control to next middleware
 */
const loggingMiddleware = function (req, res, next) {
  // Call pino HTTP logger to process request/response
  httpLogger(req, res, next);
};

// Export logging middleware for use in Express app
module.exports = loggingMiddleware;

================================================================================
FILE: costs-service/src/clients/logging_client.js
================================================================================

// Logging service client module - handles communication with external logging
service
```

```
// Import HTTP client for making requests to logging service
const axios = require("axios");
// Import configuration with logging service URL and timeout settings
const config = require("../config");

/**
 * Sends log data to external logging service via HTTP POST
 * Handles network errors gracefully without throwing
 * @param {Object} logData - Log object with level, message, timestamp, etc.
 * @returns {Promise<void>} Resolves after sending or silently fails
 */
const sendLogToService = async function (logData) {
  // Check if logging service URL is configured
  if (!config.LOGGING_SERVICE_URL) {
    // Exit early if service is not configured
    return;
  }

  try {
    // Send POST request to logging service endpoint
    await axios.post(`${config.LOGGING_SERVICE_URL}/logs`, logData, {
      // Set request timeout to prevent hanging
      timeout: config.LOGGING_SERVICE_TIMEOUT,
      // Specify JSON content type for the request
      headers: {
        "Content-Type": "application/json",
      },
    });
  } catch (error) {
    // Log error to console if service communication fails
    console.error("Failed to send log to logging service:", error.message);
  }
};

/**
 * Creates and sends a log entry to the logging service
 * Wraps log data and sends asynchronously
 * @param {Object} logData - Log object with level, message, and optional metadata
 * @returns {void} Sends log asynchronously without waiting
 */
const createLog = function (logData) {
  // Invoke async send function without awaiting to avoid blocking
  sendLogToService({
    // Spread existing log data
    ...logData,
    // Add current timestamp in ISO format
    timestamp: new Date().toISOString(),
  });
};
```

```
// Export logging client functions for use throughout application
module.exports = {
  createLog,
};

================================================================================
FILE: costs-service/src/clients/users_client.js
================================================================================

// Client for users-service communication
// Provides HTTP client interface to verify user existence
const axios = require("axios");
const config = require("../config");
const { logger } = require("../logging");

/**
 * Checks if a user exists in the users-service via HTTP request
 * Queries the /exists endpoint to verify user presence before processing costs
 * @param {number} userId - The user ID to check for existence
 * @returns {Promise<boolean>} True if user exists, false otherwise
 * @throws {Error} If HTTP request fails (connection refused, timeout, error
response)
 */
const checkUserExists = async function (userId) {
  // Build URL to users-service exists endpoint
  const baseUrl = config.USERS_SERVICE_URL;
  const url = `${baseUrl}/exists/${userId}`;

  try {
    // Make HTTP GET request with configured timeout
    const response = await axios.get(url, {
      timeout: config.USERS_SERVICE_TIMEOUT,
    });

    // Extract and return existence flag from response data
    return response.data.exists;
  } catch (error) {
    // Log error for debugging and re-throw for caller handling
    logger.error({ userId, error: error.message }, "Users-service call failed");
    throw error;
  }
};

module.exports = {
  checkUserExists,
};

================================================================================
FILE: costs-service/src/config/index.js
```

```
================================================================================

// Configuration module - loads and exports all environment settings

// Load environment variables from .env file
require("dotenv").config();

// Export configuration object with all environment variables
const env = {
  // Application environment (development, production, test)
  NODE_ENV: process.env.NODE_ENV || "development",
  // HTTP server port for listening
  PORT: process.env.PORT || 4000,
  // Log level for Pino logger (debug, info, warn, error)
  LOG_LEVEL: process.env.LOG_LEVEL || "info",
  // MongoDB connection string for database
  MONGO_URI: process.env.MONGO_URI || "mongodb://localhost:27017/app",
  // External users service base URL for validation
  USERS_SERVICE_URL:
    process.env.USERS_SERVICE_URL || "http://localhost:3000/api",
  // Request timeout for users service in milliseconds
  USERS_SERVICE_TIMEOUT: process.env.USERS_SERVICE_TIMEOUT || 3000,
  // External logging service base URL for log submission
  LOGGING_SERVICE_URL:
    process.env.LOGGING_SERVICE_URL || "http://localhost:5000/api",
  // Request timeout for logging service in milliseconds
  LOGGING_SERVICE_TIMEOUT: process.env.LOGGING_SERVICE_TIMEOUT || 3000,
};

// Export configuration for use throughout application
module.exports = env;

================================================================================
FILE: costs-service/src/config/cost_categories.js
================================================================================

// Cost categories configuration module
// Defines all valid expense categories for the costs tracking system
// Used for validation and categorization of user expenses throughout the
application

/**
 * COST_CATEGORIES object
 * Enumeration of all allowed cost categories in the system
 * Each category represents a type of expense that can be tracked
 */
const COST_CATEGORIES = {
  // Food and dining expenses (groceries, restaurants, delivery)
  FOOD: "food",
  // Healthcare and medical expenses (doctor visits, prescriptions, insurance)
```

```
  HEALTH: "health",
  // Housing and residential expenses (rent, mortgage, utilities)
  HOUSING: "housing",
  // Sports and fitness expenses (gym, equipment, classes)
  SPORTS: "sports",
  // Education and learning expenses (courses, books, training)
  EDUCATION: "education",
};

/**
 * VALID_CATEGORIES array
 * Extracted list of all valid category values
 * Used for validation and filtering in cost operations
 * Automatically generated from COST_CATEGORIES object values
 */
const VALID_CATEGORIES = Object.values(COST_CATEGORIES);

// Export configuration constants for use throughout costs-service
module.exports = {
  COST_CATEGORIES,
  VALID_CATEGORIES,
};

================================================================================
FILE: costs-service/src/db/index.js
================================================================================

// Database connection module - establishes MongoDB connection

// Import Mongoose for database operations
const mongoose = require("mongoose");
// Import configuration with database connection URI
const config = require("../config");
// Import logger for database event logging
const { logger } = require("../logging");

/**
 * Connects to MongoDB database using Mongoose
 * Logs connection status and exits process on failure
 * @returns {Promise<void>} Resolves when connected, exits on error
 */
const connectDB = async function () {
  try {
    // Attempt to establish MongoDB connection
    await mongoose.connect(config.MONGO_URI);
    // Log successful connection to database
    logger.info("MongoDB connected");
  } catch (err) {
    // Log connection error with details
    logger.error("MongoDB connection error", err);
```

```
    // Exit process immediately on connection failure
    process.exit(1);
  }
};

// Export database connection function for server startup
module.exports = { connectDB };

================================================================================
FILE: costs-service/src/db/models/cost.model.js
================================================================================

// Cost model module - Mongoose schema and model for cost entries
// Defines structure for storing individual cost records with validation rules

// Import MongoDB/Mongoose for schema definition and model creation
const mongoose = require("mongoose");
// Import valid cost categories configuration
const { VALID_CATEGORIES } = require("../../config/cost_categories");

// Define Mongoose schema for cost documents with validation rules
const costSchema = new mongoose.Schema(
  {
    // Description of the cost entry - required, trimmed string, cannot be empty
    description: {
      type: String,
      required: [true, "Field 'description' is required and must be a string"],
      trim: true,
      minlength: [1, "Field 'description' must not be empty"],
    },
    // Category of the cost - must be one of predefined valid categories (food,
education, health, housing, sports)
    category: {
      type: String,
      required: [true, "Field 'category' is required and must be a string"],
      trim: true,
      enum: {
        values: VALID_CATEGORIES,
        message: `Field 'category' must be one of: ${VALID_CATEGORIES.join(
          ", "
        )}`,
      },
    },
    // User ID - links cost to a specific user, required numeric identifier
    userid: {
      type: Number,
      required: [true, "Field 'userid' is required and must be a number"],
    },
    // Cost amount - required positive number representing the cost value
    sum: {
```

```
      type: Number,
      required: [true, "Field 'sum' is required and must be a number"],
      validate: {
        validator: function (value) {
          return value > 0;
        },
        message: "Field 'sum' must be a positive number",
      },
    },
  },
  {
    // Automatically add createdAt and updatedAt timestamp fields
    timestamps: true,
  }
);

// Create and export the Cost model for database operations
module.exports = mongoose.model("Cost", costSchema);

================================================================================
FILE: costs-service/src/db/models/monthlyReport.model.js
================================================================================

// Monthly Report model module - Mongoose schema for cached monthly cost summaries
// Stores aggregated cost data by category for each month to improve query
performance

// Import MongoDB/Mongoose for schema definition and model creation
const mongoose = require("mongoose");

// Define Mongoose schema for monthly cost report documents
const monthlyReportSchema = new mongoose.Schema(
  {
    // User ID - identifies which user this report belongs to
    userid: {
      type: Number,
      required: true,
    },
    // Year of the report - part of unique identifier (userid, year, month)
    year: {
      type: Number,
      required: true,
    },
    // Month of the report (1-12) - part of unique identifier
    month: {
      type: Number,
      required: true,
    },
    // Array of cost summaries organized by category for the month
    costs: [
```

```
      {
        // Food category costs for the month
        food: [
          {
            sum: Number,
            description: String,
            day: Number,
          },
        ],
        // Education category costs for the month
        education: [
          {
            sum: Number,
            description: String,
            day: Number,
          },
        ],
        // Health category costs for the month
        health: [
          {
            sum: Number,
            description: String,
            day: Number,
          },
        ],
        // Housing category costs for the month
        housing: [
          {
            sum: Number,
            description: String,
            day: Number,
          },
        ],
        // Sports category costs for the month
        sports: [
          {
            sum: Number,
            description: String,
            day: Number,
          },
        ],
      },
    ],
  },
  { timestamps: true }
);

// Create compound unique index on userid, year, month to prevent duplicate reports
for same period
monthlyReportSchema.index({ userid: 1, year: 1, month: 1 }, { unique: true });
```

```
// Create and export the MonthlyReport model for database operations
module.exports = mongoose.model("MonthlyReport", monthlyReportSchema);

================================================================================
FILE: costs-service/src/db/models/index.js
================================================================================

// Models aggregation module - exports all Mongoose models

const Cost = require("./cost.model");
const MonthlyReport = require("./monthlyReport.model");

module.exports = {
  Cost,
  MonthlyReport,
};

================================================================================
FILE: costs-service/src/errors/app_error.js
================================================================================

// Application error class module - defines base error for application

/**
 * Base error class for all application errors
 * Extends native JavaScript Error to add HTTP status codes
 * All custom errors should inherit from this class
 */
class AppError extends Error {
  /**
   * Constructor for creating application error instances
   * @param {string} message - Human-readable error description
   * @param {number} status - HTTP status code (default 500)
   */
  constructor(message, status) {
    // Call parent Error constructor with message
    super(message);
    // Store HTTP status code for response handling
    this.status = status;
  }
}

// Export AppError class for inheritance by specific error types
module.exports = { AppError };

================================================================================
FILE: costs-service/src/errors/duplicate_error.js
================================================================================
```

```
// Duplicate error module - handles resource conflict scenarios (HTTP 409)

const { AppError } = require("./app_error");

/**
 * DuplicateError class for resource conflict/duplication scenarios
 * Extends AppError with HTTP 409 (Conflict) status code
 */
class DuplicateError extends AppError {
  constructor(message) {
    super(message, 409);
    this.name = "DuplicateError";
  }
}

module.exports = { DuplicateError };

================================================================================
FILE: costs-service/src/errors/not_found_error.js
================================================================================

// Not found error module - handles resource not found scenarios (HTTP 404)

const { AppError } = require("./app_error");

/**
 * NotFoundError class for missing resource scenarios
 * Extends AppError with HTTP 404 (Not Found) status code
 */
class NotFoundError extends AppError {
  constructor(message) {
    super(message, 404);
    this.name = "NotFoundError";
  }
}

module.exports = { NotFoundError };

================================================================================
FILE: costs-service/src/errors/service_error.js
================================================================================

// Service error module - handles inter-service communication failures (HTTP
502/503)

const { AppError } = require("./app_error");

/**
 * ServiceError class for inter-service communication failures
 * Extends AppError with configurable HTTP status code (default 502 Bad Gateway)
```

```
 */
class ServiceError extends AppError {
  constructor(message, status = 502) {
    super(message, status);
    this.name = "ServiceError";
  }
}

module.exports = { ServiceError };

================================================================================
FILE: costs-service/src/errors/validation_error.js
================================================================================

// Validation error module - handles request validation failures (HTTP 400)

const { AppError } = require("./app_error");

/**
 * ValidationError class for data validation failures
 * Extends AppError with HTTP 400 (Bad Request) status code
 */
class ValidationError extends AppError {
  constructor(message) {
    super(message, 400);
    this.name = "ValidationError";
  }
}

module.exports = { ValidationError };

================================================================================
FILE: costs-service/src/logging/index.js
================================================================================

// Pino logger configuration with MongoDB stream
const pino = require("pino");
const loggingClient = require("../clients/logging_client");
// Import configuration for LOG_LEVEL setting
const config = require("../config");

// Map Pino numeric levels to string levels for database storage
const pinoLevelToString = {
  10: "debug", // trace
  20: "debug", // debug
  30: "info", // info
  40: "warn", // warn
  50: "error", // error
  60: "error", // fatal
};
```

```
// Custom stream to send logs to REST API via loggingClient
const customStream = {
  write: (msg) => {
    try {
      // Parse log message from pino (JSON string)
      const logObj = JSON.parse(msg);
      const level = pinoLevelToString[logObj.level] || "info";

      // Send to logging service with valid level
      loggingClient.createLog({
        level,
        message: logObj.msg,
        timestamp: new Date(logObj.time),
        method: logObj.req?.method,
        url: logObj.req?.url,
        statusCode: logObj.res?.statusCode,
        responseTime: logObj.responseTime,
      });
    } catch (err) {
      // Fallback: log error to console
      console.error("Failed to send log:", err.message);
    }
  },
};
const logger = pino(
  { level: config.LOG_LEVEL },
  pino.multistream([
    { level: config.LOG_LEVEL, stream: process.stdout },
    { level: config.LOG_LEVEL, stream: customStream },
  ])
);

module.exports = { logger };

================================================================================
FILE: costs-service/tests/unit/costs_service.test.js
================================================================================

// Costs service unit tests
// Tests for all service methods: createCost, getMonthlyReport, getUserTotalCosts
// Mocks repository and external service calls for isolated testing
// Uses Jest testing framework with mock functions for dependencies

// Mock the costs repository to avoid database calls during tests
jest.mock("../../src/app/repositories/costs_repository");
// Mock the users client to avoid inter-service calls during tests
jest.mock("../../src/clients/users_client");

// Import the service being tested
```

```
const costsService = require("../../src/app/services/costs_service");
// Import mocked dependencies for setup and verification
const costsRepository = require("../../src/app/repositories/costs_repository");
const usersClient = require("../../src/clients/users_client");
// Import database model (for reference and validation)
const Cost = require("../../src/db/models/cost.model");
// Import error classes to verify proper error handling
const { ValidationError } = require("../../src/errors/validation_error");
const { NotFoundError } = require("../../src/errors/not_found_error");
const { ServiceError } = require("../../src/errors/service_error");

describe("Costs Service", () => {
  // Reset all mocks before each test to ensure isolation
  beforeEach(() => {
    jest.clearAllMocks();
  });

  describe("createCost", () => {
    it("should create cost successfully with valid data", async () => {
      const costData = {
        description: "Lunch",
        category: "food",
        userid: 1,
        sum: 50,
      };

      const savedCost = {
        _id: "507f1f77bcf86cd799439011",
        description: "Lunch",
        category: "food",
        userid: 1,
        sum: 50,
        createdAt: new Date(),
      };

      usersClient.checkUserExists.mockResolvedValue(true);
      costsRepository.createCost.mockResolvedValue(savedCost);

      const result = await costsService.createCost(costData, "req-123");

      expect(costsRepository.createCost).toHaveBeenCalledWith({
        description: "Lunch",
        category: "food",
        userid: 1,
        sum: 50,
      });
      expect(result).toEqual(savedCost);
    });

    it("should throw ValidationError when description is missing", async () => {
```

```
      const costData = {
        category: "food",
        userid: 1,
        sum: 50,
      };

      await expect(
        costsService.createCost(costData, "req-123")
      ).rejects.toThrow(ValidationError);
    });

    it("should throw ValidationError when description is not a string", async () =>
{
      const costData = {
        description: null,
        category: "food",
        userid: 1,
        sum: 50,
      };

      await expect(
        costsService.createCost(costData, "req-123")
      ).rejects.toThrow(ValidationError);
    });

    it("should throw ValidationError when category is missing", async () => {
      const costData = {
        description: "Lunch",
        userid: 1,
        sum: 50,
      };

      await expect(
        costsService.createCost(costData, "req-123")
      ).rejects.toThrow(ValidationError);
    });

    it("should throw ValidationError when category is not a string", async () => {
      const costData = {
        description: "Lunch",
        category: null,
        userid: 1,
        sum: 50,
      };

      await expect(
        costsService.createCost(costData, "req-123")
      ).rejects.toThrow(ValidationError);
    });
```

```
it("should throw ValidationError when userid is missing", async () => {
  const costData = {
    description: "Lunch",
    category: "food",
    sum: 50,
  };

  await expect(
    costsService.createCost(costData, "req-123")
  ).rejects.toThrow(ValidationError);
});

it("should throw ValidationError when userid is not a number", async () => {
  const costData = {
    description: "Lunch",
    category: "food",
    userid: "not-a-number",
    sum: 50,
  };

  await expect(
    costsService.createCost(costData, "req-123")
  ).rejects.toThrow(ValidationError);
});

it("should throw ValidationError when sum is missing", async () => {
  const costData = {
    description: "Lunch",
    category: "food",
    userid: 1,
  };

  await expect(
    costsService.createCost(costData, "req-123")
  ).rejects.toThrow(ValidationError);
});

it("should throw ValidationError when sum is not a number", async () => {
  const costData = {
    description: "Lunch",
    category: "food",
    userid: 1,
    sum: "not-a-number",
  };

  await expect(
    costsService.createCost(costData, "req-123")
  ).rejects.toThrow(ValidationError);
});
```

```
  it("should throw ValidationError when sum is zero or negative", async () => {
    const costData = {
      description: "Lunch",
      category: "food",
      userid: 1,
      sum: 0,
    };

    await expect(
      costsService.createCost(costData, "req-123")
    ).rejects.toThrow(ValidationError);
  });

  it("should trim whitespace from description and category", async () => {
    const costData = {
      description: "  Lunch  ",
      category: "  food  ",
      userid: 1,
      sum: 50,
    };

    const savedCost = {
      _id: "507f1f77bcf86cd799439011",
      description: "Lunch",
      category: "food",
      userid: 1,
      sum: 50,
      createdAt: new Date(),
    };

    usersClient.checkUserExists.mockResolvedValue(true);
    costsRepository.createCost.mockResolvedValue(savedCost);

    await costsService.createCost(costData, "req-123");

    expect(costsRepository.createCost).toHaveBeenCalledWith({
      description: "Lunch",
      category: "food",
      userid: 1,
      sum: 50,
    });
  });
});

describe("getMonthlyReport", () => {
  it("should throw NotFoundError when user does not exist", async () => {
    usersClient.checkUserExists.mockResolvedValue(false);

    await expect(
      costsService.getMonthlyReport({
```

```
      id: 999,
      year: 2025,
      month: 6,
    })
  ).rejects.toThrow(NotFoundError);

  expect(usersClient.checkUserExists).toHaveBeenCalledWith(999);
});

it("should throw ServiceError when users service is unavailable", async () => {
  const error = new Error("connect ECONNREFUSED");
  error.code = "ECONNREFUSED";
  usersClient.checkUserExists.mockRejectedValue(error);

  await expect(
    costsService.getMonthlyReport({
      id: 1,
      year: 2025,
      month: 6,
    })
  ).rejects.toThrow(ServiceError);
});

it("should return cached report when available and user exists", async () => {
  const mockCachedReport = {
    userid: 1,
    year: 2025,
    month: 1,
    costs: [
      {
        food: [
          { sum: 50, description: "lunch", day: 5 },
          { sum: 30, description: "snack", day: 10 },
        ],
        education: [],
        health: [],
        housing: [],
        sports: [],
      },
    ],
  };

  usersClient.checkUserExists.mockResolvedValue(true);
  costsRepository.getMonthlyReportFromCache.mockResolvedValue({
    costs: mockCachedReport.costs,
  });

  const result = await costsService.getMonthlyReport(
    {
      id: 1,
```

```
      year: 2025,
      month: 1,
    },
    "req-123"
  );

  expect(costsRepository.getMonthlyReportFromCache).toHaveBeenCalledWith(
    1,
    2025,
    1
  );
  expect(result).toEqual({
    userid: 1,
    year: 2025,
    month: 1,
    costs: mockCachedReport.costs,
  });
});

it("should convert string parameters to numbers", async () => {
  const cachedData = [
    {
      food: [],
      education: [],
      health: [],
      housing: [],
      sports: [],
    },
  ];

  usersClient.checkUserExists.mockResolvedValue(true);
  costsRepository.getMonthlyReportFromCache.mockResolvedValue(null);
  costsRepository.getCostsByMonthAggregation.mockResolvedValue(cachedData);
  costsRepository.cacheMonthlyReport.mockResolvedValue({
    costs: cachedData,
  });

  const result = await costsService.getMonthlyReport(
    {
      id: "1",
      year: "2025",
      month: "6",
    },
    "req-123"
  );

  expect(costsRepository.getCostsByMonthAggregation).toHaveBeenCalledWith(
    1,
    2025,
    6
```

```
    );
    expect(result).toEqual({
      userid: 1,
      year: 2025,
      month: 6,
      costs: cachedData,
    });
  });
});

describe("getUserTotalCosts", () => {
  it("should return total costs for a user", async () => {
    costsRepository.getCostsTotalByUserId.mockResolvedValue(1500);

    const result = await costsService.getUserTotalCosts(
      { userId: 1 },
      "req-123"
    );

    expect(costsRepository.getCostsTotalByUserId).toHaveBeenCalledWith(1);
    expect(result).toEqual({
      userid: 1,
      total_costs: 1500,
    });
  });

  it("should return zero when user has no costs", async () => {
    costsRepository.getCostsTotalByUserId.mockResolvedValue(0);

    const result = await costsService.getUserTotalCosts(
      { userId: 5 },
      "req-123"
    );

    expect(result).toEqual({
      userid: 5,
      total_costs: 0,
    });
  });

  it("should throw ValidationError when userId is missing", async () => {
    await expect(
      costsService.getUserTotalCosts({}, "req-123")
    ).rejects.toThrow(ValidationError);
  });

  it("should throw ValidationError when userId is not a number", async () => {
    await expect(
      costsService.getUserTotalCosts({ userId: "not-a-number" }, "req-123")
    ).rejects.toThrow(ValidationError);
```

```
    });

    it("should convert string userId to number", async () => {
      costsRepository.getCostsTotalByUserId.mockResolvedValue(800);

      const result = await costsService.getUserTotalCosts(
        { userId: "2" },
        "req-123"
      );

      expect(costsRepository.getCostsTotalByUserId).toHaveBeenCalledWith(2);
      expect(result).toEqual({
        userid: 2,
        total_costs: 800,
      });
    });
  });
});

================================================================================
END OF COSTS SERVICE CODE DOCUMENTATION
================================================================================

================================================================================
LOGS SERVICE - COMPLETE CODE DOCUMENTATION
================================================================================

This file contains all source code files from the logs-service microservice,
organized by folder structure for easy navigation and code review.

================================================================================
FOLDER STRUCTURE
================================================================================

logs-service/
├── package.json
├── server.js
└── src/
    ├── app/
    │   ├── app.js
    │   ├── controllers/
    │   │   └── logs_controller.js
    │   ├── middlewares/
    │   │   ├── error_middleware.js
    │   │   └── logger_middleware.js
    │   ├── repositories/
    │   │   └── logs_repository.js
    │   ├── routes/
    │   │   ├── log_routes.js
    │   │   └── index.js
```

```
    │   └── services/
    │       └── log_service.js
    ├── config/
    │   └── index.js
    ├── db/
    │   ├── index.js
    │   └── models/
    │       └── log.model.js
    ├── errors/
    │   ├── app_error.js
    │   ├── duplicate_error.js
    │   ├── not_found_error.js
    │   ├── service_error.js
    │   └── validation_error.js
    └── logging/
        └── index.js
```

```
================================================================================
FILE: logs-service/server.js
================================================================================

// Server entry point for logs service - initializes database and starts
application

// Load Express application factory with routes
const createApp = require("./src/app/app");
// Import MongoDB connection initialization function
const { connectDB } = require("./src/db");
// Import logger for logging server events
const { logger } = require("./src/logging");
// Load configuration with port and service URLs
const config = require("./src/config");

// Create Express application with middleware
const app = createApp();

// Connect to MongoDB before accepting requests
connectDB().then(() => {
  // Get port from config or use default 5000
  const port = config.PORT || 5000;
  // Start listening for HTTP requests
  app.listen(port, () => {
    // Log successful server startup
    logger.info(`Server running on port ${port}`);
  });
});

================================================================================
FILE: logs-service/src/app/app.js
================================================================================
```

```
// Express application factory module - creates logs service app

// Import Express framework
const express = require("express");
// Import HTTP request logging middleware
const loggerMiddleware = require("./middlewares/logger_middleware");
// Import error handling middleware
const errorMiddleware = require("./middlewares/error_middleware");
// Import routes configuration
const routes = require("./routes");

/**
 * Factory function to create and configure Express application
 * Sets up middleware stack and API routes
 * @returns {Object} Configured Express app instance
 */
const createApp = function () {
  // Create Express application
  const app = express();
  // Apply request logging middleware
  app.use(loggerMiddleware);
  // Parse JSON request bodies
  app.use(express.json());
  // Mount API routes under /api
  app.use("/api", routes);
  // Apply error handling middleware
  app.use(errorMiddleware);

  // Return configured app
  return app;
};

// Export app factory
module.exports = createApp;

================================================================================
FILE: logs-service/src/app/controllers/logs_controller.js
================================================================================

// Logs controller module - handles HTTP requests for log retrieval and storage

// Import logs service for business logic access
const logsService = require("../services/log_service");

/**
 * Retrieves all stored logs from the database
 * Handles GET /api/logs requests
 * @param {Object} req - Express request object
 * @param {Object} res - Express response object for sending response
```

```
 * @param {Function} next - Express next middleware function for error handling
 * @returns {void} Sends 200 OK with logs array or error to next middleware
 */
const getLogs = async function (req, res, next) {
  try {
    // Fetch all logs from database through service
    const logs = await logsService.getAllLogs();
    // Return 200 OK status with logs array
    res.status(200).json(logs);
  } catch (error) {
    // Pass any errors to error handling middleware
    next(error);
  }
};

/**
 * Creates a new log entry in the database
 * Handles POST /api/logs requests with log data
 * @param {Object} req - Express request object with log data in body
 * @param {Object} res - Express response object for sending response
 * @param {Function} next - Express next middleware function for error handling
 * @returns {void} Sends 201 Created with log data or error to next
 */
const createLog = async function (req, res, next) {
  try {
    // Create log entry through service layer
    const log = await logsService.createLog(req.body);
    // Return 201 Created status with the new log object
    res.status(201).json(log);
  } catch (error) {
    // Pass validation or service errors to error middleware
    next(error);
  }
};

// Export controller functions for route handlers
module.exports = {
  getLogs,
  createLog,
};

================================================================================
FILE: logs-service/src/app/services/log_service.js
================================================================================

// Logs service - business logic layer
// Handles log validation, creation, and retrieval from database
const logsRepository = require("../repositories/logs_repository");
const { logger } = require("../../logging");
const { ValidationError } = require("../../errors/validation_error");
```

```
const Log = require("../../db/models/log.model");

/**
 * Validates log data against the Log model schema
 * Ensures all required fields are present and valid
 * @param {Object} data - Log data object with level, message, method, url,
statusCode, responseTime
 * @returns {Object} Validated log object
 * @throws {ValidationError} If log data fails schema validation
 */
const validateLogData = function (data) {
  // Create temporary log to validate against Mongoose schema
  const tempLog = new Log(data);
  const validationError = tempLog.validateSync();

  // If schema validation fails, extract and throw the first error
  if (validationError) {
    const firstErrorKey = Object.keys(validationError.errors)[0];
    const firstError = validationError.errors[firstErrorKey];
    throw new ValidationError(firstError.message);
  }

  return tempLog;
};

/**
 * Creates a new log entry in the database with validation
 * Formats and validates log data before persistence
 * @param {Object} logData - Log object with level, message, method, url,
statusCode, responseTime
 * @returns {Promise<Object>} Formatted log response with all relevant fields
 * @throws {ValidationError} If log data is invalid
 * @throws {Error} If database operation fails
 */
const createLog = async function (logData) {
  // Validate log data against schema constraints
  const validatedLog = validateLogData(logData);

  try {
    // Persist log to database with validated and formatted data
    const log = await logsRepository.createLog({
      level: validatedLog.level,
      message: validatedLog.message,
      timestamp: validatedLog.timestamp || new Date(),
      method: validatedLog.method,
      url: validatedLog.url,
      statusCode: validatedLog.statusCode,
      responseTime: validatedLog.responseTime,
    });
```

```
    // Return formatted response with all relevant fields
    return {
      _id: log._id,
      level: log.level,
      message: log.message,
      timestamp: log.timestamp,
      method: log.method,
      url: log.url,
      statusCode: log.statusCode,
      responseTime: log.responseTime,
    };
  } catch (error) {
    throw error;
  }
};

/**
 * Retrieves all logs from the database and formats them
 * Returns logs in a consistent format for API responses
 * @returns {Promise<Array>} Array of formatted log objects
 */
const getAllLogs = async function () {
  // Fetch all logs from repository (sorted by timestamp descending)
  const logs = await logsRepository.findAllLogs();

  // Format each log with consistent field structure for response
  return logs.map((log) => ({
    _id: log._id,
    level: log.level,
    message: log.message,
    timestamp: log.timestamp,
    method: log.method,
    url: log.url,
    statusCode: log.statusCode,
    responseTime: log.responseTime,
  }));
};

module.exports = {
  createLog,
  getAllLogs,
};

================================================================================
FILE: logs-service/src/app/repositories/logs_repository.js
================================================================================

// Logs repository - database access layer for log management
// Handles all database operations for application logs
const Log = require("../../db/models/log.model");
```

```
/**
 * Creates a new log entry in the database
 * Stores log information including timestamp, level, and message
 * @param {Object} logData - Log object containing timestamp, level, message,
service, user, details
 * @returns {Promise<Object>} The saved log document with _id and timestamps
 */
const createLog = async function (logData) {
  const log = new Log(logData);
  return await log.save();
};

/**
 * Retrieves all logs from the database sorted by newest first
 * @returns {Promise<Array>} Array of all log documents sorted by timestamp
descending
 */
const findAllLogs = async function () {
  return await Log.find().sort({ timestamp: -1 });
};

/**
 * Retrieves a specific log by its database ID
 * @param {string} id - The MongoDB ObjectId of the log document
 * @returns {Promise<Object|null>} The log document or null if not found
 */
const findLogById = async function (id) {
  return await Log.findById(id);
};

module.exports = {
  createLog,
  findAllLogs,
  findLogById,
};

================================================================================
FILE: logs-service/src/app/routes/log_routes.js
================================================================================

// Logs routes module - defines HTTP endpoints for log management
// Provides routes for retrieving and storing centralized logs from all services

// Import Express framework for route definition
const express = require("express");
// Create Express router instance for logs routes
const router = express.Router();
// Import logs controller with route handler functions
const logsController = require("../controllers/logs_controller");
```

```
// GET endpoint to retrieve all stored logs
// Route: GET /api/logs
// Handler: Fetches all log entries from database sorted by timestamp
router.get("/logs", logsController.getLogs);

// POST endpoint to create and store a new log entry
// Route: POST /api/logs
// Handler: Validates log data and persists to database (called by other services)
router.post("/logs", logsController.createLog);

// Export router for mounting in main routes aggregation
module.exports = router;

================================================================================
FILE: logs-service/src/app/routes/index.js
================================================================================

// Routes aggregation module - combines all API route modules for logs service
// Provides centralized routing configuration for all log-related endpoints

// Import Express framework for route definition
const express = require("express");
// Import logs-specific routes module containing all log endpoint handlers
const logsRoutes = require("./log_routes");
// Create main router instance for API routes
const router = express.Router();

// Mount logs routes at root path (all log endpoints available at /api/...)
router.use("/", logsRoutes);

// Export aggregated routes for app mounting
module.exports = router;

================================================================================
FILE: logs-service/src/app/middlewares/error_middleware.js
================================================================================

// Error handling middleware module - processes and formats error responses

// Map error types to standardized error IDs for client identification
const ERROR_ID_MAP = {
  ValidationError: "ERR_VALIDATION_001",
  NotFoundError: "ERR_NOT_FOUND_002",
  DuplicateError: "ERR_DUPLICATE_003",
  ServiceError: "ERR_SERVICE_004",
  AppError: "ERR_APP_005",
};

const getErrorId = function (err) {
```

```
  const errorType = err.constructor.name;
  return ERROR_ID_MAP[errorType] || "ERR_UNKNOWN_999";
};

const errorHandler = function (err, req, res, next) {
  const status = err.status || 500;
  const message = err.message || "Internal Server Error";
  const errorId = getErrorId(err);

  res.err = err;

  res.status(status).json({
    id: errorId,
    message: message,
  });
};

module.exports = errorHandler;

================================================================================
FILE: logs-service/src/app/middlewares/logger_middleware.js
================================================================================

// HTTP request logging middleware module - logs all HTTP requests and responses

const pinoHttp = require("pino-http");
const { logger } = require("../../logging");

const httpLogger = pinoHttp({
  logger,
  customLogLevel: function (req, res, err) {
    if (res.statusCode >= 400 && res.statusCode < 500) {
      return "warn";
    } else if (res.statusCode >= 500 || err || res.err) {
      return "error";
    }
    return "info";
  },
  customSuccessMessage: function (req, res) {
    return `${req.method} ${req.url} ${res.statusCode}`;
  },
  customErrorMessage: function (req, res, err) {
    const error = err || res.err;
    return `${req.method} ${req.url} ${res.statusCode} - ${error?.message}`;
  },
});

const loggingMiddleware = function (req, res, next) {
  httpLogger(req, res, next);
};
```

```
module.exports = loggingMiddleware;

================================================================================
FILE: logs-service/src/db/index.js
================================================================================

// Database connection module - manages MongoDB connection

// Import Mongoose ODM library
const mongoose = require("mongoose");
// Import database URI from configuration
const config = require("../config");
// Import logger for connection events
const { logger } = require("../logging");

/**
 * Establishes connection to MongoDB database
 * Logs success or exits on failure
 * @returns {Promise<void>} Resolves on connection, exits on error
 */
const connectDB = async function () {
  try {
    // Connect to MongoDB using URI from config
    await mongoose.connect(config.MONGO_URI);
    // Log successful connection
    logger.info("MongoDB connected");
  } catch (err) {
    // Log connection error
    logger.error("MongoDB connection error", err);
    // Exit process on failure
    process.exit(1);
  }
};

// Export connection function
module.exports = { connectDB };

================================================================================
FILE: logs-service/src/db/models/log.model.js
================================================================================

// Log model module - Mongoose schema for application logs
// Defines structure for storing centralized logs from all services

// Import MongoDB/Mongoose for schema definition and model creation
const mongoose = require("mongoose");

// Define Mongoose schema for log documents with validation rules
const logSchema = new mongoose.Schema(
```

```
  {
    // Log level - severity indicator: info, warn, error, or debug
    level: {
      type: String,
      required: [true, "Field 'level' is required and must be a string"],
      trim: true,
      enum: {
        values: ["info", "warn", "error", "debug"],
        message: "Field 'level' must be one of: info, warn, error, debug",
      },
    },
    // Log message - the actual log content describing what happened
    message: {
      type: String,
      required: [true, "Field 'message' is required and must be a string"],
      trim: true,
    },
    // Timestamp - when the log event occurred, defaults to current time
    timestamp: {
      type: Date,
      required: [true, "Field 'timestamp' is required"],
      default: Date.now,
      validate: {
        validator: function (value) {
          return value instanceof Date && !isNaN(value);
        },
        message: "Field 'timestamp' must be a valid date",
      },
    },
    // HTTP method - optional field for logging HTTP request method (GET, POST,
etc.)
    method: {
      type: String,
      required: false,
      trim: true,
    },
    // URL path - optional field for logging the requested endpoint
    url: {
      type: String,
      required: false,
      trim: true,
    },
    // HTTP status code - optional field for logging response status (200, 404,
500, etc.)
    statusCode: {
      type: Number,
      required: false,
    },
    // Response time in milliseconds - optional field for performance logging
    responseTime: {
```

```
      type: Number,
      required: false,
    },
  },
  {
    // Automatically add createdAt and updatedAt timestamp fields
    timestamps: true,
  }
);

// Create Log model from schema
const Log = mongoose.model("Log", logSchema);

// Export the Log model for database operations
module.exports = Log;

================================================================================
FILE: logs-service/src/config/index.js
================================================================================

// Environment configuration module for logs-service
// Loads configuration from .env file and provides fallback defaults
// All microservices use this pattern for consistent configuration management

require("dotenv").config();

/**
 * Environment configuration object
 * Aggregates all configuration needed for logs-service operation
 * Logs-service is a centralized logging microservice with minimal external
dependencies
 */
const env = {
  // Application environment (development, staging, production)
  // Controls logging verbosity and error handling behavior
  NODE_ENV: process.env.NODE_ENV || "development",

  // Port on which logs-service HTTP server listens
  // Default 5000 for local development, overridden in production
  PORT: process.env.PORT || 5000,

  // Log level for Pino logger (debug, info, warn, error)
  // Controls verbosity of application logs
  // Default info for normal operations, debug for troubleshooting
  LOG_LEVEL: process.env.LOG_LEVEL || "info",

  // MongoDB connection URI for logs database
  // Format: mongodb://[user:password@]host:port/database
  // Default points to local MongoDB instance
  // Stores all application logs from other microservices
```

```
  MONGO_URI: process.env.MONGO_URI || "mongodb://localhost:27017/app",

  // Timeout in milliseconds for internal operations
  // Used when logs-service performs any HTTP or async operations
  // Prevents hanging operations and ensures responsive error handling
  // Default 3000ms = 3 seconds
  LOGGING_SERVICE_TIMEOUT: process.env.LOGGING_SERVICE_TIMEOUT || 3000,
};

// Export configuration object for use throughout logs-service
module.exports = env;

================================================================================
FILE: logs-service/src/errors (error classes)
================================================================================

[Error classes are identical to other services - app_error.js, duplicate_error.js,
not_found_error.js, service_error.js, validation_error.js follow the same pattern
as defined in admin-service and costs-service. See those files for complete error
class implementations.]

================================================================================
FILE: logs-service/src/logging/index.js
================================================================================

// Pino logger configuration
const pino = require("pino");
// Import configuration for LOG_LEVEL setting
const config = require("../config");

// Map Pino numeric levels to string levels for database storage
const pinoLevelToString = {
  10: "debug", // trace
  20: "debug", // debug
  30: "info", // info
  40: "warn", // warn
  50: "error", // error
  60: "error", // fatal
};

// Custom stream to write directly to MongoDB (no external service call)
const customStream = {
  write: (msg) => {
    try {
      // Parse log message from pino (JSON string)
      const logObj = JSON.parse(msg);
      const level = pinoLevelToString[logObj.level] || "info";

      // In the logs service, we just log to MongoDB directly
      // No external HTTP calls to avoid infinite loops
```

```
      const logDoc = new Log({
        level,
        message: logObj.msg,
        timestamp: new Date(logObj.time),
        method: logObj.req?.method,
        url: logObj.req?.url,
        statusCode: logObj.res?.statusCode,
        responseTime: logObj.responseTime,
      });

      logDoc.save().catch((err) => {
        console.error("Failed to save log to MongoDB:", err.message);
      });
    } catch (err) {
      // Fallback: log error to console
      console.error("Failed to process log:", err.message);
    }
  },
};

const logger = pino(
  { level: config.LOG_LEVEL },
  pino.multistream([
    { level: config.LOG_LEVEL, stream: process.stdout },
  ])
);

module.exports = { logger };

================================================================================
FILE: logs-service/package.json
================================================================================

[Package.json follows the same structure as costs-service with:
- name: "hit-logs-service"
- PORT: 5000
- Same dependencies and devDependencies]

================================================================================
FILE: logs-service/tests/unit/logs.test.js
================================================================================

// Logs service unit tests
// Tests for log creation and retrieval: createLog, getAllLogs
// Validates field requirements, data types, and optional field handling
// Mocks logs repository for isolated service layer testing

// Mock the logs repository to avoid database calls during tests
jest.mock("../../src/app/repositories/logs_repository");
```

```
// Import the service being tested
const logsService = require("../../src/app/services/log_service");
// Import mocked dependencies for setup and verification
const logsRepository = require("../../src/app/repositories/logs_repository");
// Import error classes to verify proper error handling
const { ValidationError } = require("../../src/errors/validation_error");

describe("Logs Service", () => {
  // Reset all mocks before each test to ensure isolation
  beforeEach(() => {
    jest.clearAllMocks();
  });

  describe("createLog", () => {
    it("should create a log successfully with valid data", async () => {
      const logData = {
        level: "info",
        message: "Test log message",
        timestamp: new Date(),
      };

      const savedLog = {
        _id: "log123",
        level: "info",
        message: "Test log message",
        timestamp: new Date(),
      };

      logsRepository.createLog.mockResolvedValue(savedLog);

      const result = await logsService.createLog(logData);

      expect(logsRepository.createLog).toHaveBeenCalledWith(
        expect.objectContaining({
          level: "info",
          message: "Test log message",
        })
      );
      expect(result).toEqual(
        expect.objectContaining({
          _id: "log123",
          level: "info",
          message: "Test log message",
        })
      );
    });

    it("should throw ValidationError when level is missing", async () => {
      const logData = {
```

```
    message: "Test log message",
    timestamp: new Date(),
  };

  await expect(logsService.createLog(logData)).rejects.toThrow(
    ValidationError
  );

  await expect(logsService.createLog(logData)).rejects.toThrow(
    "Field 'level' is required and must be a string"
  );
});

it("should throw ValidationError when message is missing", async () => {
  const logData = {
    level: "info",
    timestamp: new Date(),
  };

  await expect(logsService.createLog(logData)).rejects.toThrow(
    ValidationError
  );

  await expect(logsService.createLog(logData)).rejects.toThrow(
    "Field 'message' is required and must be a string"
  );
});

it("should throw ValidationError when level is invalid", async () => {
  const logData = {
    level: "invalid_level",
    message: "Test log message",
    timestamp: new Date(),
  };

  await expect(logsService.createLog(logData)).rejects.toThrow(
    ValidationError
  );

  await expect(logsService.createLog(logData)).rejects.toThrow(
    "Field 'level' must be one of: info, warn, error, debug"
  );
});

it("should throw ValidationError when timestamp is invalid", async () => {
  const logData = {
    level: "info",
    message: "Test log message",
    timestamp: "invalid-date",
  };
```

```
    await expect(logsService.createLog(logData)).rejects.toThrow(
      ValidationError
    );
  });

  it("should create log with all optional fields", async () => {
    const logData = {
      level: "error",
      message: "API error occurred",
      timestamp: new Date(),
      method: "GET",
      url: "/api/users",
      statusCode: 500,
      responseTime: 150,
    };

    const savedLog = {
      _id: "log456",
      level: "error",
      message: "API error occurred",
      timestamp: new Date(),
      method: "GET",
      url: "/api/users",
      statusCode: 500,
      responseTime: 150,
    };

    logsRepository.createLog.mockResolvedValue(savedLog);

    const result = await logsService.createLog(logData);

    expect(logsRepository.createLog).toHaveBeenCalledWith(
      expect.objectContaining({
        level: "error",
        message: "API error occurred",
        method: "GET",
        url: "/api/users",
        statusCode: 500,
        responseTime: 150,
      })
    );
    expect(result.method).toBe("GET");
    expect(result.statusCode).toBe(500);
  });
});

describe("getAllLogs", () => {
  it("should return all logs", async () => {
    const logs = [
```

```
    {
      _id: "log1",
      level: "info",
      message: "Log 1",
      timestamp: new Date(),
    },
    {
      _id: "log2",
      level: "error",
      message: "Log 2",
      timestamp: new Date(),
    },
  ];

  logsRepository.findAllLogs.mockResolvedValue(logs);

  const result = await logsService.getAllLogs();

  expect(logsRepository.findAllLogs).toHaveBeenCalled();
  expect(result).toHaveLength(2);
  expect(result[0].level).toBe("info");
  expect(result[1].level).toBe("error");
});

it("should return empty array when no logs exist", async () => {
  logsRepository.findAllLogs.mockResolvedValue([]);

  const result = await logsService.getAllLogs();

  expect(result).toEqual([]);
  expect(Array.isArray(result)).toBe(true);
});

it("should sort logs by timestamp newest first", async () => {
  const now = new Date();
  const earlier = new Date(now.getTime() - 1000);
  const latest = new Date(now.getTime() + 1000);

  const logs = [
    {
      _id: "log1",
      level: "info",
      message: "Latest log",
      timestamp: latest,
    },
    {
      _id: "log2",
      level: "error",
      message: "Earlier log",
      timestamp: earlier,
```

```
        },
      ];

      logsRepository.findAllLogs.mockResolvedValue(logs);

      const result = await logsService.getAllLogs();

      expect(result[0].timestamp).toEqual(latest);
      expect(result[1].timestamp).toEqual(earlier);
    });
  });
});

================================================================================
END OF LOGS SERVICE CODE DOCUMENTATION
================================================================================

================================================================================
USERS SERVICE - COMPLETE CODE DOCUMENTATION
================================================================================

This file contains all source code files from the users-service microservice,
organized by folder structure for easy navigation and code review.

================================================================================
FOLDER STRUCTURE
================================================================================

users-service/
├── package.json
├── server.js
└── src/
    ├── app/
    │   ├── app.js
    │   ├── controllers/
    │   │   └── users_controller.js
    │   ├── middlewares/
    │   │   ├── error_middleware.js
    │   │   └── logger_middleware.js
    │   ├── repositories/
    │   │   └── users_repository.js
    │   ├── routes/
    │   │   ├── users_routes.js
    │   │   └── index.js
    │   └── services/
    │       └── users_service.js
    ├── clients/
    │   ├── costs_client.js
    │   └── logging_client.js
    ├── config/
```

```
    │   └── index.js
    ├── db/
    │   ├── index.js
    │   └── models/
    │       └── user.model.js
    ├── errors/
    │   ├── app_error.js
    │   ├── duplicate_error.js
    │   ├── not_found_error.js
    │   ├── service_error.js
    │   └── validation_error.js
    └── logging/
        └── index.js
```

```
================================================================================
FILE: users-service/server.js
================================================================================

// Server entry point for users service - initializes database and starts
application

// Load Express application factory
const createApp = require("./src/app/app");
// Import database connection function
const { connectDB } = require("./src/db");
// Import logger for server events
const { logger } = require("./src/logging");
// Load environment configuration
const config = require("./src/config");

// Create Express application with middleware
const app = createApp();

// Connect to MongoDB before accepting requests
connectDB().then(() => {
  // Get port from config or use default 3000
  const port = config.PORT || 3000;
  // Start listening for HTTP requests
  app.listen(port, () => {
    // Log successful server startup
    logger.info(`Server running on port ${port}`);
  });
});

================================================================================
FILE: users-service/src/app/app.js
================================================================================

// Express application factory module - creates users service app
```

```
// Import Express framework for app creation
const express = require("express");
// Import HTTP request logging middleware
const loggerMiddleware = require("./middlewares/logger_middleware");
// Import error handling middleware
const errorMiddleware = require("./middlewares/error_middleware");
// Import routes configuration
const routes = require("./routes");

/**
 * Factory function to create and configure Express application
 * Sets up all middleware and routes
 * @returns {Object} Configured Express app instance
 */
const createApp = function () {
  // Create Express app instance
  const app = express();
  // Apply request logging middleware
  app.use(loggerMiddleware);
  // Parse JSON request bodies
  app.use(express.json());
  // Mount API routes
  app.use("/api", routes);
  // Apply error handling middleware
  app.use(errorMiddleware);

  // Return configured app
  return app;
};

// Export app factory
module.exports = createApp;

================================================================================
FILE: users-service/src/app/controllers/users_controller.js
================================================================================

// Users controller module - handles HTTP requests for user management endpoints

// Import users service for business logic
const usersService = require("../services/users_service");

/**
 * Creates a new user in the database
 * Handles POST /api/add requests
 * @param {Object} req - Express request object with user data in body
 * @param {Object} res - Express response object for sending response
 * @param {Function} next - Express next middleware function for error handling
 * @returns {void} Sends 201 Created with user data or error to next
 */
```

```
const addUser = async function (req, res, next) {
  try {
    // Validate and create user through service
    const user = await usersService.addUser(req.body);
    // Return 201 Created status with new user object
    res.status(201).json(user);
  } catch (error) {
    // Pass errors to error middleware
    next(error);
  }
};

/**
 * Retrieves all users from the database
 * Handles GET /api/users requests
 * @param {Object} req - Express request object
 * @param {Object} res - Express response object for sending response
 * @param {Function} next - Express next middleware function for error handling
 * @returns {void} Sends 200 OK with users array or error to next
 */
const getUsers = async function (req, res, next) {
  try {
    // Fetch all users from service layer
    const users = await usersService.getAllUsers();
    // Return 200 OK status with users array
    res.status(200).json(users);
  } catch (error) {
    // Pass errors to error middleware
    next(error);
  }
};

/**
 * Retrieves a specific user by ID with cost information
 * Handles GET /api/users/:id requests
 * @param {Object} req - Express request object with user ID in params
 * @param {Object} res - Express response object for sending response
 * @param {Function} next - Express next middleware function for error handling
 * @returns {void} Sends 200 OK with user data or error to next
 */
const getUserById = async function (req, res, next) {
  try {
    // Fetch user by ID from service layer
    const user = await usersService.getUserById(req.params.id);
    // Return 200 OK status with user object
    res.status(200).json(user);
  } catch (error) {
    // Pass errors to error middleware
    next(error);
  }
```

```
};

/**
 * Checks if a user exists in the database
 * Handles GET /api/exists/:id requests
 * @param {Object} req - Express request object with user ID in params
 * @param {Object} res - Express response object for sending response
 * @param {Function} next - Express next middleware function for error handling
 * @returns {void} Sends 200 OK with exists status or error to next
 */
const checkUserExists = async function (req, res, next) {
  try {
    // Check if user exists in database
    const result = await usersService.checkUserExists(req.params.id);
    // Return 200 OK status with exists result
    res.status(200).json(result);
  } catch (error) {
    // Pass errors to error middleware
    next(error);
  }
};

// Export controller functions for route handlers
module.exports = {
  addUser,
  getUsers,
  getUserById,
  checkUserExists,
};

================================================================================
FILE: users-service/src/app/services/users_service.js (Part 1)
================================================================================

// Users service - business logic layer
// Handles user validation, CRUD operations, and cost integration via external
service
const usersRepository = require("../repositories/users_repository");
const costsClient = require("../../clients/costs_client");
const { logger } = require("../../logging");
const { ValidationError } = require("../../errors/validation_error");
const { NotFoundError } = require("../../errors/not_found_error");
const { DuplicateError } = require("../../errors/duplicate_error");
const { ServiceError } = require("../../errors/service_error");
const User = require("../../db/models/user.model");

/**
 * Validates user data against the User model schema
 * Ensures all required fields are present and valid
 * @param {Object} data - User data object with id, first_name, last_name, birthday
```

```
 * @returns {Object} Validated user object
 * @throws {ValidationError} If user data fails schema validation
 */
const validateUserData = function (data) {
  // Create temporary user to validate against Mongoose schema
  const tempUser = new User(data);
  const validationError = tempUser.validateSync();

  // If schema validation fails, extract and throw the first error
  if (validationError) {
    const firstErrorKey = Object.keys(validationError.errors)[0];
    const firstError = validationError.errors[firstErrorKey];
    throw new ValidationError(firstError.message);
  }

  return tempUser;
};

/**
 * Creates a new user with validation and duplicate checking
 * Validates user data, checks for existing user, and persists to database
 * @param {Object} userData - User object with id, first_name, last_name, birthday
 * @returns {Promise<Object>} Formatted user response with all fields
 * @throws {ValidationError} If user data is invalid
 * @throws {DuplicateError} If user with same id already exists
 */
const addUser = async function (userData) {
  // Validate user data against schema constraints
  const validatedUser = validateUserData(userData);

  // Check if user with this id already exists to prevent duplicates
  const exists = await usersRepository.checkUserExists(validatedUser.id);
  if (exists) {
    throw new DuplicateError(`User with id ${validatedUser.id} already exists`);
  }

  try {
    // Persist user to database with validated data
    const user = await usersRepository.createUser({
      id: validatedUser.id,
      first_name: validatedUser.first_name,
      last_name: validatedUser.last_name,
      birthday: validatedUser.birthday,
    });

    // Log successful user creation for audit trail
    logger.info({ userId: user.id }, "User created");

    // Return formatted response with all relevant fields
    return {
```

```
      id: user.id,
      first_name: user.first_name,
      last_name: user.last_name,
      birthday: user.birthday,
    };
  } catch (error) {
    // Handle MongoDB duplicate key error (mongoDB-specific error code)
    if (error.code === 11000) {
      throw new DuplicateError(
        `User with id ${validatedUser.id} already exists`
      );
    }
    throw error;
  }
};

/**
 * Retrieves all users from the database and formats them
 * Returns users in a consistent format for API responses
 * @returns {Promise<Array>} Array of formatted user objects
 */
const getAllUsers = async function () {
  // Fetch all users from repository
  const users = await usersRepository.findAllUsers();

  // Format each user with consistent field structure for response
  return users.map((user) => ({
    id: user.id,
    first_name: user.first_name,
    last_name: user.last_name,
    birthday: user.birthday,
  }));
};

/**
 * Retrieves a specific user with their total costs from costs service
 * Integrates user profile data with cost information via external service call
 * @param {string|number} id - User ID to retrieve
 * @returns {Promise<Object>} User object with total_costs field
 * @throws {ValidationError} If id is not a valid number
 * @throws {NotFoundError} If user doesn't exist
 * @throws {ServiceError} If costs service is unavailable
 */
const getUserById = async function (id) {
  // Parse and validate user ID parameter
  const userId = parseInt(id, 10);

  if (isNaN(userId)) {
    throw new ValidationError("Invalid user id");
  }
```

```
  // Query user from repository by ID
  const user = await usersRepository.findUserById(userId);

  if (!user) {
    throw new NotFoundError(`User with id ${userId} not found`);
  }

  // Fetch total costs from external costs service
  let totalCosts;
  try {
    totalCosts = await costsClient.getUserTotalCosts(userId);
  } catch (error) {
    // Handle service communication errors (connection refused, timeout, 5xx)
    if (
      error.code === "ECONNREFUSED" ||
      error.code === "ETIMEDOUT" ||
      error.response?.status >= 500
    ) {
      throw new ServiceError("Costs service unavailable", 503);
    } else if (error.response) {
      // Handle HTTP error responses from costs service
      throw new ServiceError(
        `Costs service error: ${error.response.statusText}`,
        502
      );
    } else {
      // Handle unexpected errors during service call
      throw new ServiceError("Failed to fetch costs data", 502);
    }
  }

  // Return formatted response with user and costs data
  return {
    id: user.id,
    first_name: user.first_name,
    last_name: user.last_name,
    total_costs: totalCosts,
  };
};

/**
 * Checks if a user with the given ID exists in the system
 * Validates ID format and queries repository for existence
 * @param {string|number} id - User ID to check for existence
 * @returns {Promise<Object>} Object with exists boolean and userId
 * @throws {ValidationError} If id is not a valid number
 */
const checkUserExists = async function (id) {
  // Parse and validate user ID parameter
```

```
  const userId = parseInt(id, 10);

  if (isNaN(userId)) {
    throw new ValidationError("Invalid user id");
  }

  // Query repository to check if user exists
  const exists = await usersRepository.checkUserExists(userId);

  // Return formatted response with existence flag
  return {
    exists,
    userId,
  };
};

module.exports = {
  addUser,
  getAllUsers,
  getUserById,
  checkUserExists,
};

================================================================================
FILE: users-service/src/app/repositories/users_repository.js
================================================================================

// Users repository - database access layer for user management
// Handles all database operations for user profiles and authentication
const User = require("../../db/models/user.model");

/**
 * Creates a new user in the database
 * Stores user profile information (name, email, id)
 * @param {Object} userData - User object containing id, name, email properties
 * @returns {Promise<Object>} The saved user document with MongoDB _id and
timestamps
 */
const createUser = async function (userData) {
  const user = new User(userData);
  return await user.save();
};

/**
 * Retrieves a user by their unique identifier
 * @param {number} id - The user's unique identifier (numeric ID)
 * @returns {Promise<Object|null>} The user document or null if not found
 */
const findUserById = async function (id) {
  return await User.findOne({ id });
```

```
};

/**
 * Retrieves all users from the database
 * @returns {Promise<Array>} Array of all user documents in the system
 */
const findAllUsers = async function () {
  return await User.find({});
};

/**
 * Checks whether a user with the given ID exists in the database
 * Used for validation before creating or accessing user resources
 * @param {number} id - The user ID to check for existence
 * @returns {Promise<boolean>} True if user exists, false otherwise
 */
const checkUserExists = async function (id) {
  const count = await User.countDocuments({ id });
  return count > 0;
};

module.exports = {
  createUser,
  findUserById,
  findAllUsers,
  checkUserExists,
};

================================================================================
FILE: users-service/src/app/routes/users_routes.js
================================================================================

// Users routes module - defines HTTP endpoints for user management
// Provides routes for creating users, retrieving profiles, and checking existence

// Import Express framework for route definition
const express = require("express");
// Import users controller with route handler functions
const usersController = require("../controllers/users_controller");

// Create Express router instance for users routes
const router = express.Router();

// POST endpoint to create a new user
// Route: POST /api/add
// Handler: Validates user data, checks for duplicates, and persists to database
router.post("/add", usersController.addUser);

// GET endpoint to retrieve all users
// Route: GET /api/users
```

```
// Handler: Fetches all user profiles from database
router.get("/users", usersController.getUsers);

// GET endpoint to retrieve a specific user by ID with cost information
// Route: GET /api/users/:id
// Handler: Returns user profile with total costs from costs service
router.get("/users/:id", usersController.getUserById);

// GET endpoint to check if a user exists
// Route: GET /api/exists/:id
// Handler: Validates user existence for inter-service communication
router.get("/exists/:id", usersController.checkUserExists);

// Export router for mounting in main routes aggregation
module.exports = router;

================================================================================
FILE: users-service/src/app/routes/index.js
================================================================================

// Routes aggregation module - combines all API route modules for users service
// Provides centralized routing configuration for all user-related endpoints

// Import Express framework for route definition
const express = require("express");
// Import users-specific routes module containing all user endpoint handlers
const usersRoutes = require("./users_routes");
// Create main router instance for API routes
const router = express.Router();

// Mount users routes at root path (all user endpoints available at /api/...)
router.use("/", usersRoutes);

// Export aggregated routes for app mounting
module.exports = router;

================================================================================
FILE: users-service/src/app/middlewares/error_middleware.js
================================================================================

// Error handling middleware module - processes and formats error responses

const ERROR_ID_MAP = {
  ValidationError: "ERR_VALIDATION_001",
  NotFoundError: "ERR_NOT_FOUND_002",
  DuplicateError: "ERR_DUPLICATE_003",
  ServiceError: "ERR_SERVICE_004",
  AppError: "ERR_APP_005",
};
```

```
const getErrorId = function (err) {
  const errorType = err.constructor.name;
  return ERROR_ID_MAP[errorType] || "ERR_UNKNOWN_999";
};

const errorHandler = function (err, req, res, next) {
  const status = err.status || 500;
  const message = err.message || "Internal Server Error";
  const errorId = getErrorId(err);

  res.err = err;

  res.status(status).json({
    id: errorId,
    message: message,
  });
};

module.exports = errorHandler;

================================================================================
FILE: users-service/src/app/middlewares/logger_middleware.js
================================================================================

// HTTP request logging middleware module - logs all HTTP requests and responses

const pinoHttp = require("pino-http");
const { logger } = require("../../logging");

const httpLogger = pinoHttp({
  logger,
  customLogLevel: function (req, res, err) {
    if (res.statusCode >= 400 && res.statusCode < 500) {
      return "warn";
    } else if (res.statusCode >= 500 || err || res.err) {
      return "error";
    }
    return "info";
  },
  customSuccessMessage: function (req, res) {
    return `${req.method} ${req.url} ${res.statusCode}`;
  },
  customErrorMessage: function (req, res, err) {
    const error = err || res.err;
    return `${req.method} ${req.url} ${res.statusCode} - ${error?.message}`;
  },
});

const loggingMiddleware = function (req, res, next) {
  httpLogger(req, res, next);
```

```
};

module.exports = loggingMiddleware;

================================================================================
FILE: users-service/src/clients/costs_client.js
================================================================================

// Client for costs-service communication
// Provides HTTP client interface to retrieve user cost totals
const axios = require("axios");
const config = require("../config");
const { logger } = require("../logging");

/**
 * Retrieves total costs for a specific user from the costs-service
 * Queries the /user-total endpoint to get aggregated cost sum
 * @param {number} userId - The user ID to retrieve total costs for
 * @returns {Promise<number>} Total sum of all costs for the user
 * @throws {Error} If HTTP request fails (connection refused, timeout, error
response)
 */
const getUserTotalCosts = async function (userId) {
  // Build URL to costs-service user-total endpoint
  const baseUrl = config.COSTS_SERVICE_URL;
  const url = `${baseUrl}/user-total`;

  try {
    // Make HTTP GET request with userId parameter and configured timeout
    const response = await axios.get(url, {
      params: { userId },
      timeout: config.COSTS_SERVICE_TIMEOUT,
    });

    // Extract and return total costs from response data
    return response.data.total_costs;
  } catch (error) {
    // Log error for debugging and re-throw for caller handling
    logger.error({ userId, error: error.message }, "Costs-service call failed");
    throw error;
  }
};

module.exports = {
  getUserTotalCosts,
};

================================================================================
FILE: users-service/src/clients/logging_client.js
================================================================================
```

```
// Logging service client module - handles communication with external logging
service

// Import HTTP client for making requests to logging service
const axios = require("axios");
// Import configuration with logging service URL and timeout settings
const config = require("../config");

/**
 * Sends log data to external logging service via HTTP POST
 * Handles network errors gracefully without throwing
 * @param {Object} logData - Log object with level, message, timestamp, etc.
 * @returns {Promise<void>} Resolves after sending or silently fails
 */
const sendLogToService = async function (logData) {
  // Check if logging service URL is configured
  if (!config.LOGGING_SERVICE_URL) {
    // Exit early if service is not configured
    return;
  }

  try {
    // Send POST request to logging service endpoint
    await axios.post(`${config.LOGGING_SERVICE_URL}/logs`, logData, {
      // Set request timeout to prevent hanging
      timeout: config.LOGGING_SERVICE_TIMEOUT,
      // Specify JSON content type for the request
      headers: {
        "Content-Type": "application/json",
      },
    });
  } catch (error) {
    // Log error to console if service communication fails
    console.error("Failed to send log to logging service:", error.message);
  }
};

/**
 * Creates and sends a log entry to the logging service
 * Wraps log data and sends asynchronously
 * @param {Object} logData - Log object with level, message, and optional metadata
 * @returns {void} Sends log asynchronously without waiting
 */
const createLog = function (logData) {
  // Invoke async send function without awaiting to avoid blocking
  sendLogToService({
    // Spread existing log data
    ...logData,
    // Add current timestamp in ISO format
```

```
    timestamp: new Date().toISOString(),
  });
};

// Export logging client functions for use throughout application
module.exports = {
  createLog,
};
```

```
================================================================================
FILE: users-service/src/config/index.js
================================================================================
```

```
// Environment configuration module for users-service
// Loads configuration from .env file and provides fallback defaults
// All microservices use this pattern for consistent configuration management

require("dotenv").config();

/**
 * Environment configuration object
 * Aggregates all configuration needed for users-service operation
 * Includes database, service discovery URLs, and timeout settings
 */
const env = {
  // Application environment (development, staging, production)
  // Controls logging verbosity and error handling behavior
  NODE_ENV: process.env.NODE_ENV || "development",

  // Port on which users-service HTTP server listens
  // Default 3000 for local development, overridden in production
  PORT: process.env.PORT || 3000,

  // Log level for Pino logger (debug, info, warn, error)
  // Controls verbosity of application logs
  // Default info for normal operations, debug for troubleshooting
  LOG_LEVEL: process.env.LOG_LEVEL || "info",

  // MongoDB connection URI for users database
  // Format: mongodb://[user:password@]host:port/database
  // Default points to local MongoDB instance
  MONGO_URI: process.env.MONGO_URI || "mongodb://localhost:27017/app",

  // Base URL for costs-service API calls
  // Used when users-service needs to fetch user costs from costs-service
  // Example: http://localhost:4000/api
  COSTS_SERVICE_URL:
    process.env.COSTS_SERVICE_URL || "http://localhost:4000/api",

  // Timeout in milliseconds for costs-service HTTP requests
```

```
  // Prevents hanging requests if costs-service is slow or unresponsive
  // Default 3000ms = 3 seconds
  COSTS_SERVICE_TIMEOUT: process.env.COSTS_SERVICE_TIMEOUT || 3000,

  // Base URL for logging-service API calls
  // Used to send application logs to centralized logging service
  // Example: http://localhost:5000/api
  LOGGING_SERVICE_URL:
    process.env.LOGGING_SERVICE_URL || "http://localhost:5000/api",

  // Timeout in milliseconds for logging-service HTTP requests
  // Prevents request timeouts when sending logs
  // Default 3000ms = 3 seconds
  LOGGING_SERVICE_TIMEOUT: process.env.LOGGING_SERVICE_TIMEOUT || 3000,
};

// Export configuration object for use throughout users-service
module.exports = env;

================================================================================
FILE: users-service/src/db/index.js
================================================================================

// Database connection module - manages MongoDB connections

// Import Mongoose ODM library
const mongoose = require("mongoose");
// Import MongoDB URI from configuration
const config = require("../config");
// Import logger for connection logging
const { logger } = require("../logging");

/**
 * Establishes connection to MongoDB database
 * Logs connection status and exits on failure
 * @returns {Promise<void>} Resolves on success, exits process on failure
 */
const connectDB = async function () {
  try {
    // Connect to MongoDB using configured URI
    await mongoose.connect(config.MONGO_URI);
    // Log successful connection
    logger.info("MongoDB connected");
  } catch (err) {
    // Log connection error
    logger.error("MongoDB connection error", err);
    // Exit process on failure
    process.exit(1);
  }
};
```

```
// Export database connection function
module.exports = { connectDB };

================================================================================
FILE: users-service/src/db/models/user.model.js
================================================================================

// User model module - Mongoose schema and model for user profiles
// Defines structure for storing user information with validation rules

// Import MongoDB/Mongoose for schema definition and model creation
const mongoose = require("mongoose");

// Define Mongoose schema for user documents with validation rules
const userSchema = new mongoose.Schema(
  {
    // User ID - unique identifier for the user, must be numeric, indexed for fast
lookups
    id: {
      type: Number,
      required: [true, "Field 'id' is required and must be a number"],
      unique: true,
      index: true,
    },
    // User's first name - required string field, whitespace trimmed
    first_name: {
      type: String,
      required: [true, "Field 'first_name' is required and must be a string"],
      trim: true,
    },
    // User's last name - required string field, whitespace trimmed
    last_name: {
      type: String,
      required: [true, "Field 'last_name' is required and must be a string"],
      trim: true,
    },
    // User's birthday - required valid date, validated to ensure proper date
format
    birthday: {
      type: Date,
      required: [true, "Field 'birthday' is required"],
      validate: {
        validator: function (value) {
          return value instanceof Date && !isNaN(value);
        },
        message: "Field 'birthday' must be a valid date",
      },
    },
  },
```

```
  {
    // Automatically add createdAt and updatedAt timestamp fields
    timestamps: true,
  }
);

// Create User model from schema
const User = mongoose.model("User", userSchema);

// Export the User model for database operations
module.exports = User;
```

```
================================================================================
FILE: users-service/src/errors (error classes)
================================================================================

[Error classes follow the same pattern as defined in admin-service and
costs-service. See those files for complete implementations of:
- app_error.js
- duplicate_error.js
- not_found_error.js
- service_error.js
- validation_error.js]

================================================================================
FILE: users-service/src/logging/index.js
================================================================================
```

```
// Pino logger configuration
const pino = require("pino");
const loggingClient = require("../clients/logging_client");
// Import configuration for LOG_LEVEL setting
const config = require("../config");

// Map Pino numeric levels to string levels for database storage
const pinoLevelToString = {
  10: "debug",
  20: "debug",
  30: "info",
  40: "warn",
  50: "error",
  60: "error",
};

// Custom stream to send logs to logging service
const customStream = {
  write: (msg) => {
    try {
      const logObj = JSON.parse(msg);
      const level = pinoLevelToString[logObj.level] || "info";
```

```
      loggingClient.createLog({
        level,
        message: logObj.msg,
        timestamp: new Date(logObj.time),
        method: logObj.req?.method,
        url: logObj.req?.url,
        statusCode: logObj.res?.statusCode,
        responseTime: logObj.responseTime,
      });
    } catch (err) {
      console.error("Failed to send log:", err.message);
    }
  },
};

const logger = pino(
  { level: config.LOG_LEVEL },
  pino.multistream([
    { level: config.LOG_LEVEL, stream: process.stdout },
    { level: config.LOG_LEVEL, stream: customStream },
  ])
);

module.exports = { logger };

================================================================================
FILE: users-service/package.json
================================================================================

{
  "name": "hit-users-service",
  "version": "1.0.0",
  "description": "Users microservice",
  "main": "server.js",
  "scripts": {
    "test": "cross-env NODE_ENV=test jest",
    "test:watch": "cross-env NODE_ENV=test jest --watch",
    "start": "node server.js",
    "dev": "cross-env PORT=3000 nodemon server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "commonjs",
  "dependencies": {
    "axios": "^1.7.9",
    "cross-env": "^7.0.3",
    "dotenv": "^17.2.3",
    "express": "^5.2.1",
```

```
    "mongoose": "^9.1.1",
    "pino": "^10.1.0",
    "pino-http": "^11.0.0"
  },
  "devDependencies": {
    "cross-env": "^10.1.0",
    "jest": "^30.2.0",
    "nodemon": "^3.1.11",
    "pino-pretty": "^13.1.3",
    "supertest": "^7.2.2"
  }
}

================================================================================
FILE: users-service/tests/unit/users_controller.test.js
================================================================================

// Users controller unit tests
// Tests for all controller endpoints: addUser, getUsers, getUserById,
checkUserExists
// Validates HTTP request/response handling and error propagation
// Mocks service layer for isolated controller testing

// Mock the users service to avoid service layer execution during tests
jest.mock("../../src/app/services/users_service");

// Import the controller being tested
const usersController = require("../../src/app/controllers/users_controller");
// Import mocked dependency for setup and verification
const usersService = require("../../src/app/services/users_service");
// Import error classes to verify proper error handling
const { ValidationError } = require("../../src/errors/validation_error");
const { NotFoundError } = require("../../src/errors/not_found_error");
const { DuplicateError } = require("../../src/errors/duplicate_error");
const { ServiceError } = require("../../src/errors/service_error");

describe("Users Controller", () => {
  let req;
  let res;
  let next;

  beforeEach(() => {
    req = {
      body: {},
      params: {},
      headers: {},
      id: "test-request-id",
    };
    res = {
      status: jest.fn().mockReturnThis(),
```

```
    json: jest.fn(),
  };
  next = jest.fn();
  jest.clearAllMocks();
});

describe("POST /api/add", () => {
  it("should create a new user successfully", async () => {
    const userData = {
      id: 1,
      first_name: "John",
      last_name: "Doe",
      birthday: "1990-01-01",
    };

    req.body = userData;

    const expectedUser = {
      id: 1,
      first_name: "John",
      last_name: "Doe",
      birthday: new Date("1990-01-01"),
    };

    usersService.addUser.mockResolvedValue(expectedUser);

    await usersController.addUser(req, res, next);

    expect(usersService.addUser).toHaveBeenCalledWith(userData);
    expect(res.status).toHaveBeenCalledWith(201);
    expect(res.json).toHaveBeenCalledWith(expectedUser);
    expect(next).not.toHaveBeenCalled();
  });

  it("should return 400 when required fields are missing", async () => {
    req.body = {
      id: 1,
      first_name: "John",
    };

    const validationError = new ValidationError(
      "Field 'last_name' is required and must be a string"
    );

    usersService.addUser.mockRejectedValue(validationError);

    await usersController.addUser(req, res, next);

    expect(next).toHaveBeenCalledWith(validationError);
    expect(res.status).not.toHaveBeenCalled();
```

```
  });

  it("should return 409 when user id already exists", async () => {
    req.body = {
      id: 1,
      first_name: "John",
      last_name: "Doe",
      birthday: "1990-01-01",
    };

    const duplicateError = new DuplicateError(
      "User with id 1 already exists"
    );

    usersService.addUser.mockRejectedValue(duplicateError);

    await usersController.addUser(req, res, next);

    expect(next).toHaveBeenCalledWith(duplicateError);
    expect(res.status).not.toHaveBeenCalled();
  });
});

describe("GET /api/users", () => {
  it("should return array of users", async () => {
    const users = [
      {
        id: 1,
        first_name: "John",
        last_name: "Doe",
        birthday: new Date("1990-01-01"),
      },
      {
        id: 2,
        first_name: "Jane",
        last_name: "Smith",
        birthday: new Date("1992-05-15"),
      },
    ];

    usersService.getAllUsers.mockResolvedValue(users);

    await usersController.getUsers(req, res, next);

    expect(usersService.getAllUsers).toHaveBeenCalledWith();
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.json).toHaveBeenCalledWith(users);
    expect(next).not.toHaveBeenCalled();
  });
});
```

```
  describe("GET /api/users/:id", () => {
    it("should return user with total_costs when costs-service responds", async ()
=> {
      req.params.id = "1";

      const userWithCosts = {
        id: 1,
        first_name: "John",
        last_name: "Doe",
        birthday: new Date("1990-01-01"),
        total_costs: 150.5,
      };

      usersService.getUserById.mockResolvedValue(userWithCosts);

      await usersController.getUserById(req, res, next);

      expect(usersService.getUserById).toHaveBeenCalledWith("1");
      expect(res.status).toHaveBeenCalledWith(200);
      expect(res.json).toHaveBeenCalledWith(userWithCosts);
      expect(next).not.toHaveBeenCalled();
    });

    it("should return 404 when user not found", async () => {
      req.params.id = "999";

      const notFoundError = new NotFoundError("User with id 999 not found");

      usersService.getUserById.mockRejectedValue(notFoundError);

      await usersController.getUserById(req, res, next);

      expect(next).toHaveBeenCalledWith(notFoundError);
      expect(res.status).not.toHaveBeenCalled();
    });

    it("should return 503 when costs-service is unavailable", async () => {
      req.params.id = "1";

      const serviceError = new ServiceError("Costs service unavailable", 503);

      usersService.getUserById.mockRejectedValue(serviceError);

      await usersController.getUserById(req, res, next);

      expect(next).toHaveBeenCalledWith(serviceError);
      expect(res.status).not.toHaveBeenCalled();
    });
  });
```

```
});

================================================================================
FILE: users-service/tests/unit/users_service.test.js
================================================================================

// Users service unit tests
// Tests for all service methods: addUser, getAllUsers, getUserById
// Validates required fields, duplicate detection, and integration with
costs-service
// Mocks repository and external service calls for isolated testing
// Uses Jest testing framework with mock functions for dependencies

// Mock the users repository to avoid database calls during tests
jest.mock("../../src/app/repositories/users_repository");
// Mock the costs client to avoid inter-service calls during tests
jest.mock("../../src/clients/costs_client");

// Import the service being tested
const usersService = require("../../src/app/services/users_service");
// Import mocked dependencies for setup and verification
const usersRepository = require("../../src/app/repositories/users_repository");
const costsClient = require("../../src/clients/costs_client");
// Import error classes to verify proper error handling
const { ValidationError } = require("../../src/errors/validation_error");
const { NotFoundError } = require("../../src/errors/not_found_error");
const { DuplicateError } = require("../../src/errors/duplicate_error");
const { ServiceError } = require("../../src/errors/service_error");

describe("Users Service", () => {
  beforeEach(() => {
    jest.clearAllMocks();
  });

  describe("addUser", () => {
    it("should create user successfully with valid data", async () => {
      const userData = {
        id: 1,
        first_name: "John",
        last_name: "Doe",
        birthday: "1990-01-01",
      };

      const savedUser = {
        id: 1,
        first_name: "John",
        last_name: "Doe",
        birthday: new Date("1990-01-01"),
      };
```

```
  usersRepository.checkUserExists.mockResolvedValue(false);
  usersRepository.createUser.mockResolvedValue(savedUser);

  const result = await usersService.addUser(userData);

  expect(usersRepository.checkUserExists).toHaveBeenCalledWith(1);
  expect(usersRepository.createUser).toHaveBeenCalledWith({
    id: 1,
    first_name: "John",
    last_name: "Doe",
    birthday: expect.any(Date),
  });
  expect(result).toEqual({
    id: 1,
    first_name: "John",
    last_name: "Doe",
    birthday: expect.any(Date),
  });
});

it("should throw ValidationError when id is missing", async () => {
  const userData = {
    first_name: "John",
    last_name: "Doe",
    birthday: "1990-01-01",
  };

  await expect(usersService.addUser(userData)).rejects.toThrow(
    ValidationError
  );

  await expect(usersService.addUser(userData)).rejects.toThrow(
    "Field 'id' is required and must be a number"
  );
});

it("should throw ValidationError when first_name is missing", async () => {
  const userData = {
    id: 1,
    last_name: "Doe",
    birthday: "1990-01-01",
  };

  await expect(usersService.addUser(userData)).rejects.toThrow(
    ValidationError
  );

  await expect(usersService.addUser(userData)).rejects.toThrow(
    "Field 'first_name' is required and must be a string"
  );
```

```
});

it("should throw ValidationError when last_name is missing", async () => {
  const userData = {
    id: 1,
    first_name: "John",
    birthday: "1990-01-01",
  };

  await expect(usersService.addUser(userData)).rejects.toThrow(
    ValidationError
  );

  await expect(usersService.addUser(userData)).rejects.toThrow(
    "Field 'last_name' is required and must be a string"
  );
});

it("should throw ValidationError when birthday is missing", async () => {
  const userData = {
    id: 1,
    first_name: "John",
    last_name: "Doe",
  };

  await expect(usersService.addUser(userData)).rejects.toThrow(
    ValidationError
  );

  await expect(usersService.addUser(userData)).rejects.toThrow(
    "Field 'birthday' is required"
  );
});

it("should throw ValidationError when birthday is invalid", async () => {
  const userData = {
    id: 1,
    first_name: "John",
    last_name: "Doe",
    birthday: "invalid-date",
  };

  await expect(usersService.addUser(userData)).rejects.toThrow(
    ValidationError
  );
});

it("should throw DuplicateError when user id already exists", async () => {
  const userData = {
    id: 1,
```

```
        first_name: "John",
        last_name: "Doe",
        birthday: "1990-01-01",
      };

      usersRepository.checkUserExists.mockResolvedValue(true);

      await expect(usersService.addUser(userData)).rejects.toThrow(
        DuplicateError
      );
    });

    it("should throw DuplicateError when mongo returns duplicate key error", async
() => {
      const userData = {
        id: 1,
        first_name: "John",
        last_name: "Doe",
        birthday: "1990-01-01",
      };

      usersRepository.checkUserExists.mockResolvedValue(false);

      const dbError = new Error("Duplicate key");
      dbError.code = 11000;
      usersRepository.createUser.mockRejectedValue(dbError);

      await expect(usersService.addUser(userData)).rejects.toThrow(
        DuplicateError
      );
    });
  });

  describe("getAllUsers", () => {
    it("should return all users", async () => {
      const users = [
        {
          id: 1,
          first_name: "John",
          last_name: "Doe",
          birthday: new Date("1990-01-01"),
        },
        {
          id: 2,
          first_name: "Jane",
          last_name: "Smith",
          birthday: new Date("1992-05-15"),
        },
      ];
```

```
      usersRepository.findAllUsers.mockResolvedValue(users);

      const result = await usersService.getAllUsers();

      expect(usersRepository.findAllUsers).toHaveBeenCalled();
      expect(result).toEqual([
        {
          id: 1,
          first_name: "John",
          last_name: "Doe",
          birthday: new Date("1990-01-01"),
        },
        {
          id: 2,
          first_name: "Jane",
          last_name: "Smith",
          birthday: new Date("1992-05-15"),
        },
      ]);
    });

    it("should return empty array when no users exist", async () => {
      usersRepository.findAllUsers.mockResolvedValue([]);

      const result = await usersService.getAllUsers();

      expect(result).toEqual([]);
    });
  });

  describe("getUserById", () => {
    it("should return user with total_costs when costs-service responds", async ()
=> {
      const user = {
        id: 1,
        first_name: "John",
        last_name: "Doe",
        birthday: new Date("1990-01-01"),
      };

      usersRepository.findUserById.mockResolvedValue(user);
      costsClient.getUserTotalCosts.mockResolvedValue(150.5);

      const result = await usersService.getUserById("1");

      expect(usersRepository.findUserById).toHaveBeenCalledWith(1);
      expect(costsClient.getUserTotalCosts).toHaveBeenCalledWith(1);
      expect(result).toEqual({
        id: 1,
        first_name: "John",
```

```
        last_name: "Doe",
        total_costs: 150.5,
      });
    });

    it("should throw NotFoundError when user does not exist", async () => {
      usersRepository.findUserById.mockResolvedValue(null);

      await expect(usersService.getUserById("999")).rejects.toThrow(
        NotFoundError
      );
    });

    it("should throw ValidationError when id is not a number", async () => {
      await expect(usersService.getUserById("abc")).rejects.toThrow(
        ValidationError
      );
    });

    it("should throw ServiceError with 503 when costs-service is unavailable",
async () => {
      const user = {
        id: 1,
        first_name: "John",
        last_name: "Doe",
        birthday: new Date("1990-01-01"),
      };

      usersRepository.findUserById.mockResolvedValue(user);

      const error = new Error("connect ECONNREFUSED");
      error.code = "ECONNREFUSED";
      costsClient.getUserTotalCosts.mockRejectedValue(error);

      await expect(usersService.getUserById("1")).rejects.toThrow(ServiceError);

      try {
        await usersService.getUserById("1");
      } catch (err) {
        expect(err.status).toBe(503);
      }
    });

    it("should throw ServiceError with 503 when costs-service times out", async ()
=> {
      const user = {
        id: 1,
        first_name: "John",
        last_name: "Doe",
        birthday: new Date("1990-01-01"),
```

```
      };

      usersRepository.findUserById.mockResolvedValue(user);

      const error = new Error("timeout");
      error.code = "ETIMEDOUT";
      costsClient.getUserTotalCosts.mockRejectedValue(error);

      await expect(usersService.getUserById("1")).rejects.toThrow(ServiceError);

      try {
        await usersService.getUserById("1");
      } catch (err) {
        expect(err.status).toBe(503);
      }
    });

    it("should throw ServiceError with 502 when costs-service returns error
response", async () => {
      const user = {
        id: 1,
        first_name: "John",
        last_name: "Doe",
        birthday: new Date("1990-01-01"),
      };

      usersRepository.findUserById.mockResolvedValue(user);

      const error = new Error("Request failed");
      error.response = {
        status: 400,
        statusText: "Bad Request",
      };
      costsClient.getUserTotalCosts.mockRejectedValue(error);

      await expect(usersService.getUserById("1")).rejects.toThrow(ServiceError);

      try {
        await usersService.getUserById("1");
      } catch (err) {
        expect(err.status).toBe(502);
      }
    });
  });
});

================================================================================
END OF USERS SERVICE CODE DOCUMENTATION
================================================================================
```