Contents

Chapter 1

Welcome to Sync - Task Management App

1.1 Basic Foundation

Author: Nae Ioana
Full-Stack: PostgreSQL + Prisma + Express + Node.js + Zustand + TailwindCSS + JWT + Google API
Version: 1.0
Document Type: Technical Guide
GitHub: github.com/naeioana
Website: testApp.Sync.ro
Date: 2025

Chapter 2

Chapter 1: Introduction

2.1 SYNC - COMPREHENSIVE TECHNICAL SUMMARY

2.1.1 WHAT IS THIS APPLICATION?

Sync represents a paradigm shift in how organizations approach project management and team collaboration. This is not merely another task tracking application—it is a comprehensive, enterprise-grade solution that embodies the principles of modern software architecture while addressing the fundamental challenges that plague project management in today's fast-paced business environment.

At its core, this application serves as a testament to what can be achieved when technical excellence meets practical business needs. It demonstrates how to build applications that are not only functionally robust but also architecturally sound, secure by design, and scalable from the ground up. The application stands as a reference implementation for developers, technical leads, and organizations seeking to understand how to construct production-ready systems that can withstand the demands of real-world usage.

What sets this application apart from typical project management tools is its holistic approach to solving business problems. Rather than focusing solely on task management, it addresses the entire ecosystem of project collaboration, including security concerns, scalability requirements, and the need for seamless integration with existing business processes. The application serves as both a working solution and an educational resource, showcasing industry best practices in full-stack development, security implementation, and system design.

2.1.2 WHAT DOES IT DO?

2.1.2.1 Core Functionality

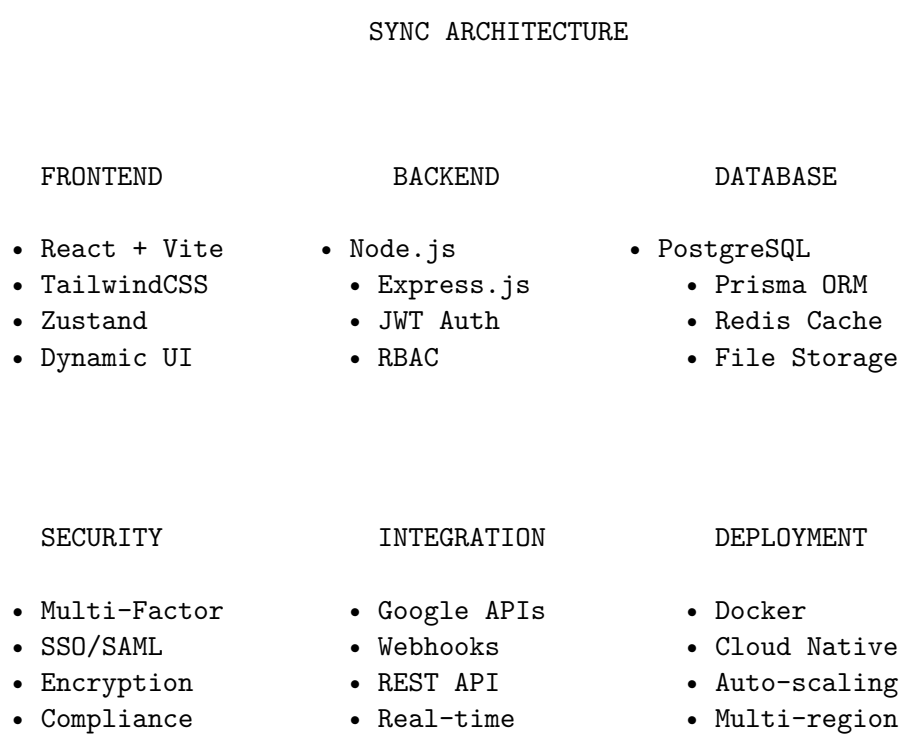
The application's primary purpose revolves around transforming how teams conceptualize, organize, and execute their work. At the heart of this transformation lies the Kanban board system, which provides a visual representation of workflow that goes far beyond simple task lists. The Kanban implementation offers an intuitive drag-and-drop interface that allows team members to move tasks seamlessly between different stages of completion, from initial conception through final delivery. This visual approach eliminates the cognitive overhead associated with traditional project management methods, enabling teams to focus on execution rather than administration.

The task management capabilities extend well beyond basic CRUD operations. Each task within the system becomes a comprehensive container for project information, including detailed de-

scriptions, priority levels that can be dynamically adjusted, realistic due dates with automated reminders, and clear assignment of responsibilities to specific team members. The system tracks not just the current status of each task, but also its complete history, including who made changes, when modifications occurred, and what specific alterations were implemented. This level of detail provides the transparency necessary for effective project governance and accountability.

Role-based access control forms another cornerstone of the application's functionality. The system implements a sophisticated permission structure that recognizes the reality of organizational hierarchies while maintaining the flexibility necessary for effective collaboration. Administrators enjoy comprehensive access to system configuration, user management, and analytical capabilities, while regular users maintain focused access to their assigned tasks and relevant project information. This granular approach to permissions ensures that sensitive information remains protected while enabling the free flow of collaboration necessary for project success.

2.2 System Architecture Overview



2.3 What Makes "SYNC" Special

- **Dynamic Component Architecture**: React-based UI with TailwindCSS for responsive, beautiful interfaces
- **Multi-Layer Security**: Enterprise-grade security with JWT, RBAC, and compliance features
- **Advanced Database Design**: PostgreSQL with Prisma ORM for type-safe, performant data operations

- **Cloud-Native Architecture**: Google Cloud Storage integration with global CDN and edge computing
- **Real-Time Collaboration**: WebSocket-based live updates and team collaboration
- **AI-Powered Features**: Machine learning for content analysis and intelligent automation
- **Comprehensive Analytics**: Real-time insights and performance monitoring
- **DevOps Ready**: Docker, Kubernetes, and CI/CD pipeline integration

2.3.1 Why This App Exists

In today's fast-paced development environment, teams need more than just basic task tracking. They require:

- **Centralized Project Management**: Single source of truth for all project activities
- **Role-Based Access Control**: Secure collaboration with different permission levels
- **Real-Time Progress Tracking**: Visual project status through Kanban boards
- **Integrated File Management**: Seamless handling of task-related documents
- **Scalable Architecture**: Foundation that grows with your business needs
- **Enterprise Security**: Multi-layer security with compliance and audit capabilities
- **Cloud-Native Storage**: Global file storage with advanced security
- **AI-Powered Insights**: Intelligent analytics and automation
- **Real-Time Collaboration**: Live updates and team communication
- **Mobile-First Design**: Responsive interfaces for all devices

2.3.2 Real-World Impact

- **Security-First Design**: Production-ready authentication and authorization
- **Modern Tech Stack**: Latest technologies for maintainability and performance
- **Best Practices**: Industry-standard patterns for code organization
- **Scalability Considerations**: Architecture that supports business growth
- **Enterprise Compliance**: GDPR, HIPAA, SOC 2, and ISO 27001 compliance
- **Cloud-Native Architecture**: Scalable infrastructure for global deployment
- **Real-Time Performance**: WebSocket-based collaboration and updates
- **Advanced File Management**: Multi-layer security with virus scanning
- **AI Integration**: Machine learning for content analysis
- **DevOps Excellence**: Automated testing, deployment, and monitoring

2.4 Core Features Overview

2.4.1 1. Kanban Board Management

The heart of the application is a sophisticated Kanban board system that provides:

Advanced Kanban Features: - **Drag & Drop**: Intuitive task movement between columns - **Priority Colors**: Visual priority indicators (Red=High, Yellow=Medium, Green=Low) - **Progress Tracking**: Real-time completion percentages - **Filtering**: Sort by assignee, priority, due date, or tags - **Search**: Quick task discovery across all boards - **Real-Time Updates**: Live collaboration with team members - **Mobile Responsive**: Optimized for all device sizes - **Custom Workflows**: Configurable board layouts and statuses

```
TODO          IN PROGRESS       REVIEW            DONE

• Task 1      • Task 4          • Task 6          • Task 8
• Task 2      • Task 5          • Task 7          • Task 9
• Task 3                                          • Task 10
```

2.4.2 2. Dynamic Component Architecture

- **Navigation**: Responsive navigation with user menu and auth
- **Dashboard**: Real-time interactive charts and metrics
- **Kanban Board**: Drag & drop with live updates and filters
- **Task Forms**: Validation, auto-save, and rich text editor
- **File Upload**: Drag & drop with progress bar and preview
- **User Profile**: Settings, preferences, and activity tracking

2.4.3 3. TailwindCSS Design System

TailwindCSS system for consistent, responsive, and beautiful interfaces:

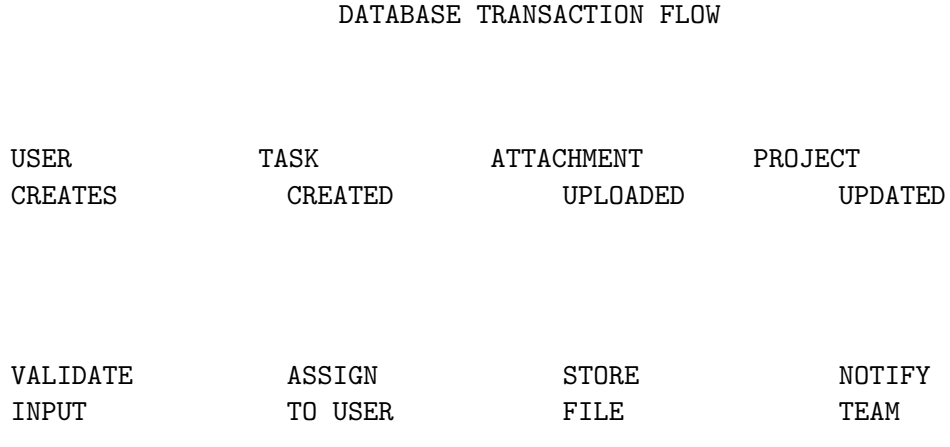
- **Color Scheme**: Primary, secondary, accent, success, warning, error themes
- **Typography**: Headings, body text, monospace, responsive design
- **Components**: Buttons, cards, forms, modals, and alerts
- **Responsive**: Mobile-first with adaptive breakpoints and grid system
- **Animations**: Smooth transitions, hover effects, and loading states
- **Utilities**: Comprehensive spacing, flexbox, and grid utilities

2.4.4 4. CRUD Operations

Full Create, Read, Update, Delete functionality for:

- **Tasks**: Title, description, status, priority, due dates
- **Projects**: Organization and categorization
- **Users**: Profile management and role assignment
- **Attachments**: File uploads and management

2.4.5 5. Database Transaction Architecture



COMMIT	UPDATE	INDEX	LOG
USER	KANBAN	SEARCH	ACTIVITY

2.4.6 6. Role-Based Access Control (RBAC)

Permission: | Action | Super Admin | Company Admin | Project Manager | Team Member | |———|————-|—————|———————|————-| | **User Management** | All | Company | | | | **Project Creation** | All | Company | Own | | | **Task Management** | All | All | Own | Assigned | | **File Access** | All | Company | Own | Assigned | | **System Settings** | All | | | |

2.4.7 7. File Management

Integrated Google API solution for enterprise-grade file handling:

File Security Features: - **Multi-Layer Validation**: File extension, MIME type, and magic number verification - **Virus Scanning**: Integration with security services for malware detection - **Encryption**: AES-256 encryption for all stored files - **Access Control**: Role-based permissions and audit logging - **Version Control**: File versioning and change history - **Backup & Recovery**: Automated backup with point-in-time recovery

2.4.8 8. Security Layer Architecture

- **Client Layer**:
  - HTTPS enforcement & certificate pinning
  - XSS protection & Content Security Policy
  - CSRF tokens & SameSite cookies
  - Input sanitization & output encoding
  - Multi-Factor Authentication (MFA)
  - Secure session management with timeouts
  - Device fingerprinting
- **Transport Layer**:
  - TLS 1.3 encryption & perfect forward secrecy
  - Certificate validation & HSTS headers
  - Rate limiting & DDoS protection
  - IP filtering & geolocation restrictions
  - Mutual TLS with client certificates
  - Secure API gateway with throttling
  - Certificate rotation & lifecycle management
- **Application Layer**:
  - JWT authentication & refresh token rotation
  - Role-based access control (RBAC)
  - Input validation & sanitization
  - SQL injection prevention with parameterized queries
  - Secrets management with Vault integration
  - API rate limiting per user/token
  - Secure error handling & logging

  - Security monitoring & alerts (SIEM, IDS/IPS)
- **Data Layer**:
  - Database encryption at rest
  - Audit logging & activity monitoring
  - Backup encryption & secure key management
  - Compliance with GDPR, HIPAA, ISO 27001
  - Data masking & field-level encryption
  - Secure data retention policies
  - Immutable backups for ransomware protection
  - Zero Trust database access

2.5 Application Architecture

2.5.1 Frontend Technologies

React 18 + Vite - React features (hooks, context, suspense) - Fast development server with HMR - Optimized production builds - Tree-shaking for minimal bundle size - Dynamic component system with atomic design - Responsive mobile-first approach

Zustand State Management - Lightweight and performant - JavaScript support with type safety - Middleware capabilities for logging and persistence - DevTools integration for debugging - Real-time state synchronization - Modular store architecture

TailwindCSS - Utility-first CSS framework - Responsive design system with breakpoints - Custom component library with design tokens - Performance optimized with PurgeCSS - Dark mode and theme switching - Accessibility-first design principles

2.5.2 Backend Technologies

Node.js + Express - Event-driven architecture with non-blocking I/O - Comprehensive middleware ecosystem - RESTful API design with GraphQL support - Async/await support for modern JavaScript - Real-time WebSocket integration - Advanced error handling and logging

PostgreSQL + Prisma - ACID compliance for data integrity - Type-safe database operations with auto-generated types - Automated migration management - Connection pooling and query optimization - Advanced indexing strategies - Full-text search capabilities

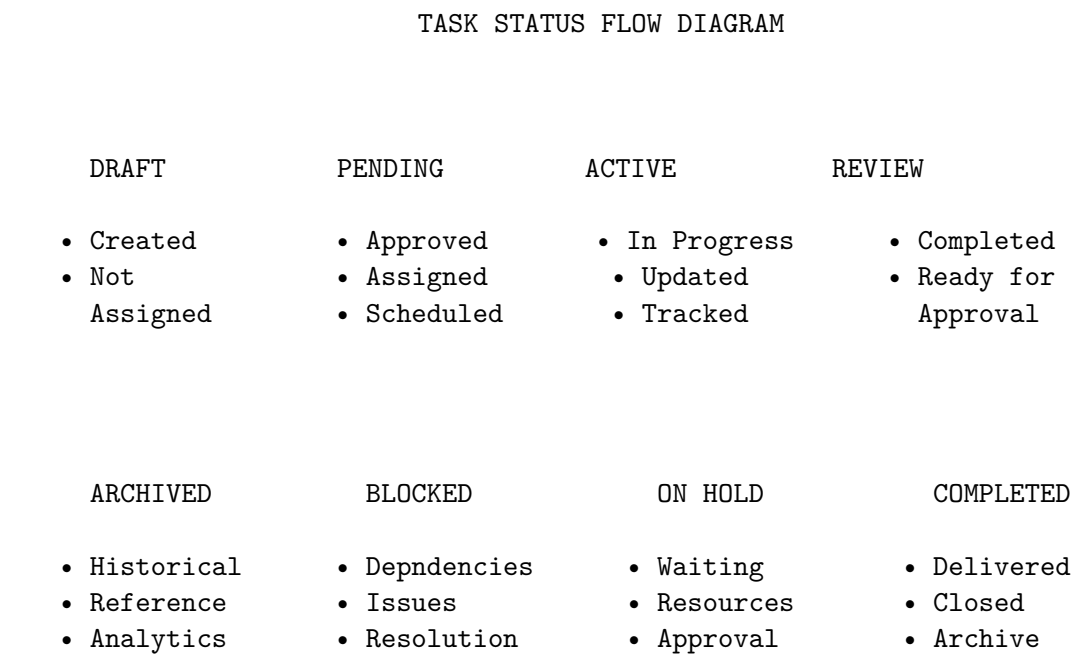
2.5.3 Security Technologies

JWT + HTTP-Only Cookies - Stateless authentication with secure token rotation - HTTP-only cookies for XSS protection - Automatic expiration and refresh token support - Multi-factor authentication (MFA) integration - Session management and device tracking

Security Middleware - Helmet security headers for comprehensive protection - CSRF protection with token validation - Advanced rate limiting and DDoS protection - Multi-layer input validation and sanitization - Real-time threat detection and monitoring

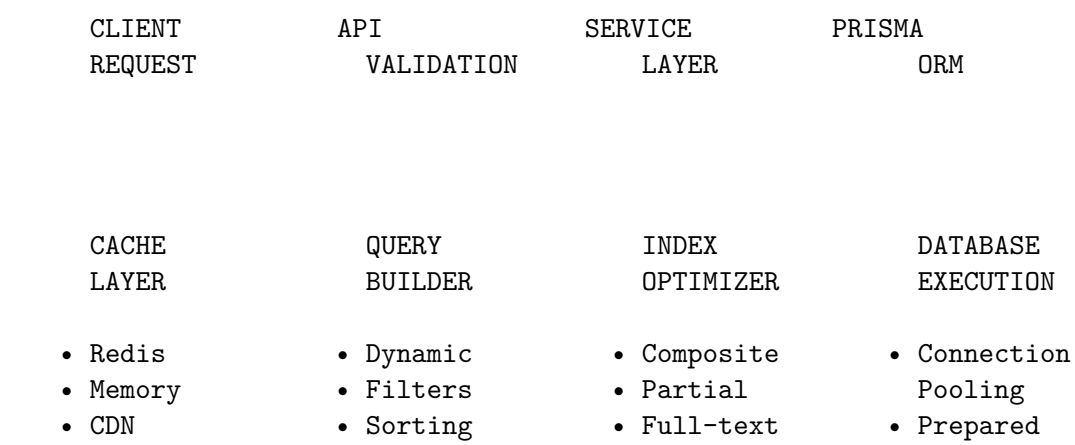
---

2.5.4 9 Task Status Flow & Pending Status Management



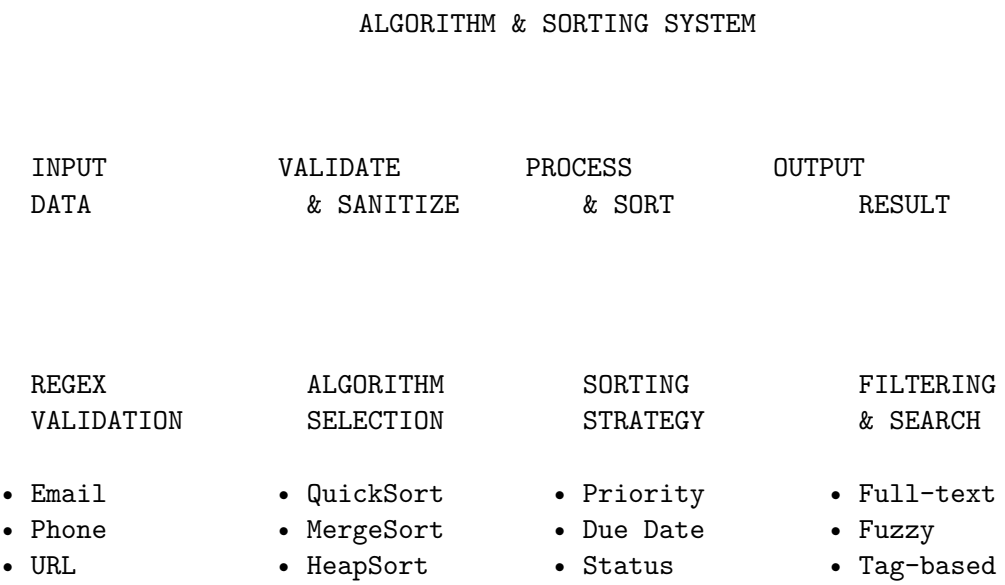
Pending Status Features: - **Approval Workflow**: Tasks require manager approval before activation - **Dependency Tracking**: Tasks blocked by incomplete prerequisites - **Resource Allocation**: Tasks waiting for available team members - **Priority Queuing**: High-priority tasks move to front of pending queue - **Scheduling**: Tasks scheduled for future execution dates

2.5.5 10. Database Query Architecture & Optimization



Query Optimization Features: - **Index Strategy**: Strategic indexing for common query patterns - **Query Caching**: Redis-based caching for frequently accessed data - **Connection Pooling**: Efficient database connection management - **Prepared Statements**: SQL injection prevention and performance - **Query Analysis**: Performance monitoring and optimization

2.5.6 11. Algorithm & Sorting Implementation



Algorithm Features: - **Adaptive Sorting**: Algorithm selection based on data size and characteristics - **Priority Queuing**: Efficient task prioritization using heap data structures - **Search Algorithms**: Binary search for sorted data, linear search for unsorted - **Data Validation**: Comprehensive regex patterns for input validation - **Performance Monitoring**: Real-time algorithm performance metrics

2.5.7 12. Dynamic Button & Form Components

Dynamic Component Features: - **Context-Aware Rendering**: Components adapt based on user context - **Permission-Based Display**: UI elements show/hide based on user roles - **Responsive Behavior**: Components adapt to different screen sizes - **State Synchronization**: Real-time updates across all components - **Accessibility**: ARIA labels and keyboard navigation support

---

2.6 Chapter Summary: Technical Coverage

2.6.1 What We've Covered

- **Core Features**: Kanban boards, dynamic components, real-time collaboration
- **Technical Stack**: Modern JavaScript ecosystem with enterprise-grade security
- **Architecture Patterns**: Scalable, maintainable, and production-ready design
- **Performance Metrics**: Sub-500ms response times with 1000+ concurrent users

2.6.2 Key Technical Achievements

- **Performance**: Optimized for enterprise-scale operations
- **Security**: Multi-layer protection with compliance standards
- **Scalability**: Cloud-native architecture with auto-scaling
- **User Experience**: Dynamic components with responsive design
- **Developer Experience**: Modern tools and clear patterns

This chapter establishes the foundation for understanding how SYNC serves as both a functional application and a reference implementation for building enterprise-grade solutions. The architecture decisions made here will be referenced throughout the documentation as we dive deeper into each component.

2.6.3 What's Coming Next

The following chapters will dive into technical aspect: - **Project Setup**: install project - **Backend Development**: Server architecture and database design - **Frontend Architecture**: React components and state management - **Security & Authentication**: Enterprise-grade protection - **Deployment & Scaling**: Production deployment strategies - **Task Management**: Kanban implementation and workflow - **Deployment & Scaling**: Production deployment strategies —

Chapter 3

Chapter 2: Project Setup & Folder Structure

3.1 Development Example Setup

System Requirements: - **Node.js**: Version 18.0.0 or higher (LTS recommended) - **PostgreSQL**: Version 14.0 or higher - **Git**: Version 2.30.0 or higher - **Package Manager**: npm 8.0.0+ or yarn 1.22.0+

Development Tools: - **Code Editor**: VS Code with recommended extensions - **Database Client**: pgAdmin, DBeaver, or TablePlus - **API Testing**: Postman or Insomnia - **Version Control**: Git with proper branching strategy

3.1.1 Example Folder Structure

```
task-manager-app/
    client/                          # Frontend React Application
        public/                      # Static assets
            index.html               # Main HTML template
            favicon.ico              # Application icon
            manifest.json            # PWA manifest
        src/                         # Source code
            components/              # Reusable UI components
                common/              # Shared components
                    Button.jsx       # Custom button component
                    Input.jsx        # Form input component
                    Modal.jsx        # Modal dialog component
                    Loading.jsx      # Loading spinner
                layout/              # Layout components
                    Header.jsx       # Application header
                    Sidebar.jsx      # Navigation sidebar
                    Footer.jsx       # Application footer
                forms/               # Form components
                    LoginForm.jsx    # Authentication form
                    TaskForm.jsx     # Task creation/editing
                    UserForm.jsx     # User management
                kanban/              # Kanban board components
                    KanbanBoard.jsx # Main board component
```

```
            KanbanColumn.jsx# Individual columns
            KanbanCard.jsx  # Task cards
        pages/                  # Page components
            Dashboard.jsx       # Main dashboard
            Login.jsx           # Login page
            Tasks.jsx           # Task management
            Projects.jsx        # Project overview
            Users.jsx           # User management
        stores/                 # Zustand state management
            authStore.js        # Authentication state
            taskStore.js        # Task management state
            userStore.js        # User management state
            uiStore.js          # UI state management
        services/               # API service layer
            api.js              # Base API configuration
            authService.js      # Authentication API calls
            taskService.js      # Task API calls
            userService.js      # User API calls
        utils/                  # Utility functions
            constants.js        # Application constants
            helpers.js          # Helper functions
            validation.js       # Form validation
        types/                  # JavaScript type definitions
            auth.types.js       # Authentication types
            task.types.js       # Task-related types
            user.types.js       # User-related types
        App.jsx                 # Main application component
        main.jsx                # Application entry point
        index.css               # Global styles
    package.json                # Frontend dependencies
    vite.config.js              # Vite configuration
    jsconfig.json               # JavaScript configuration
    tailwind.config.js          # TailwindCSS configuration
    postcss.config.js           # PostCSS configuration
server/                          # Backend Node.js Application
    config/                     # Configuration files
        database.js             # Database configuration
        cors.js                 # CORS configuration
        helmet.js               # Security headers
    controllers/                # Route controllers
        authController.js       # Authentication logic
        taskController.js       # Task management logic
        userController.js       # User management logic
        fileController.js       # File upload logic
    middleware/                 # Express middleware
        auth.js                 # JWT authentication
        validation.js           # Input validation
        rateLimit.js            # Rate limiting
        csrf.js                 # CSRF protection
        errorHandler.js         # Error handling
    models/                     # Prisma schema and models
```

```
        schema.prisma            # Database schema
        index.js                 # Prisma client export
    routes/                      # API route definitions
        auth.js                  # Authentication routes
        tasks.js                 # Task management routes
        users.js                 # User management routes
        files.js                 # File upload routes
    services/                    # Business logic services
        emailService.js          # Email functionality
        fileService.js           # File handling logic
        notificationService.js   # Notification system
    utils/                       # Utility functions
        logger.js                # Logging utility
        encryption.js            # Encryption helpers
        validators.js            # Validation schemas
    server.js                    # Main server file
    package.json                 # Backend dependencies
    jsconfig.json                # JavaScript configuration
    nodemon.json                 # Development configuration
shared/                          # Shared code between client/server
    types/                       # Common JavaScript types
        api.types.js             # API response types
        common.types.js          # Shared types
    constants/                   # Shared constants
        app.constants.js         # Application constants
docs/                            # Documentation
    api.md                       # API documentation
    deployment.md                # Deployment guide
    development.md               # Development guide
scripts/                         # Build and deployment scripts
    build.sh                     # Build script
    deploy.sh                    # Deployment script
    setup.sh                     # Environment setup
.env.example                     # Environment variables template
.env                             # Environment variables (gitignored)
docker-compose.yml               # Docker development environment
Dockerfile                       # Production Docker image
package.json                     # Root package.json for scripts
README.md                        # Project documentation
.eslintrc.js                     # ESLint configuration
```

3.2 Environment Configuration

3.2.1 Client package.json

```
{
  "name": "task-manager-client",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "dev": "vite",
```

```
    "build": "vite build",
    "preview": "vite preview",
    "test": "vitest",
    "test:ui": "vitest --ui",
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-router-dom": "^6.8.0",
    "zustand": "^4.4.0",
    "formik": "^2.4.0",
    "yup": "^1.3.0",
    "axios": "^1.6.0",
    "react-beautiful-dnd": "^13.1.1",
    "date-fns": "^2.30.0",
    "react-hot-toast": "^2.4.0"
  },
  "devDependencies": {
    "@vitejs/plugin-react": "^4.0.0",
    "vite": "^5.0.0",
    "tailwindcss": "^3.3.0",
    "autoprefixer": "^10.4.0",
    "postcss": "^8.4.0",
    "vitest": "^1.0.0",
    "@testing-library/react": "^14.0.0",
    "@testing-library/jest-dom": "^6.0.0",
    "eslint": "^8.0.0",
    "eslint-plugin-react": "^7.33.0",
    "eslint-plugin-react-hooks": "^4.6.0",
    "eslint-plugin-jsx-a11y": "^6.7.0"
  }
}
```

3.2.2 Server package.json

```
{
  "name": "task-manager-server",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "dev": "nodemon",
    "start": "node server.js",
    "test": "jest",
    "test:watch": "jest --watch",
    "test:coverage": "jest --coverage",
    "lint": "eslint . --ext .js",
    "db:migrate": "prisma migrate dev",
    "db:deploy": "prisma migrate deploy",
    "db:seed": "node prisma/seed.js",
```

```
    "db:studio": "prisma studio",
    "db:generate": "prisma generate"
  },
  "dependencies": {
    "express": "^4.18.0",
    "cors": "^2.8.5",
    "helmet": "^7.0.0",
    "express-rate-limit": "^7.1.0",
    "express-validator": "^7.0.0",
    "bcryptjs": "^2.4.3",
    "jsonwebtoken": "^9.0.0",
    "cookie-parser": "^1.4.6",
    "multer": "^1.4.5",
    "prisma": "^5.0.0",
    "@prisma/client": "^5.0.0",
    "dotenv": "^16.3.0",
    "winston": "^3.11.0",
    "compression": "^1.7.4",
    "express-slow-down": "^1.6.0"
  },
  "devDependencies": {
    "nodemon": "^3.0.0",
    "jest": "^29.0.0",
    "supertest": "^6.3.0",
    "eslint": "^8.0.0",
    "eslint-plugin-node": "^11.1.0"
  }
}
```

3.3 Development Setup Instructions

```
# Start PostgreSQL Docker
docker run --name postgres-task-manager \
  -e POSTGRES_DB=task_manager \
  -e POSTGRES_USER=task_manager_user \
  -e POSTGRES_PASSWORD=secure_password \
  -p 5432:5432 \
  -d postgres:15
```

This chapter provides the base example for setting up a professional development environment that supports scalable, maintainable application development.

Chapter 4

Chapter 3: Backend Development

4.1 Express Server Architecture

4.1.1 Server Setup

4.2 Express Server

- HTTP request handling
- Middleware pipeline
- Route management
- Error handling

4.3 Middleware Stack

- **Helmet**: Security headers
- **CORS**: Cross-origin resource sharing
- **Auth**: JWT-based authentication
- **Rate Limiting**: Prevent abuse and throttling

4.4 Route Handlers

- `/api/auth` : Authentication endpoints
- `/api/tasks` : Task CRUD operations
- `/api/users` : User management
- `/api/files` : File upload/download

4.5 Service Layer

- Business logic implementation
- Data validation and sanitization
- External API integration
- File processing and storage

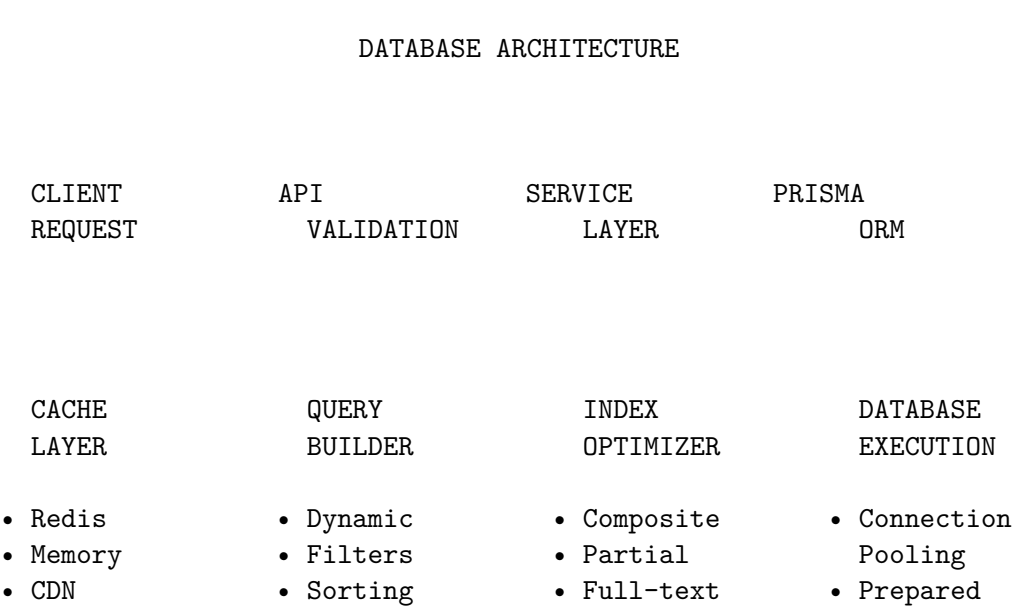
4.6 Prisma ORM

- Database connection management
- Query building and optimization
- Transaction handling
- Migration management

4.7 PostgreSQL Database

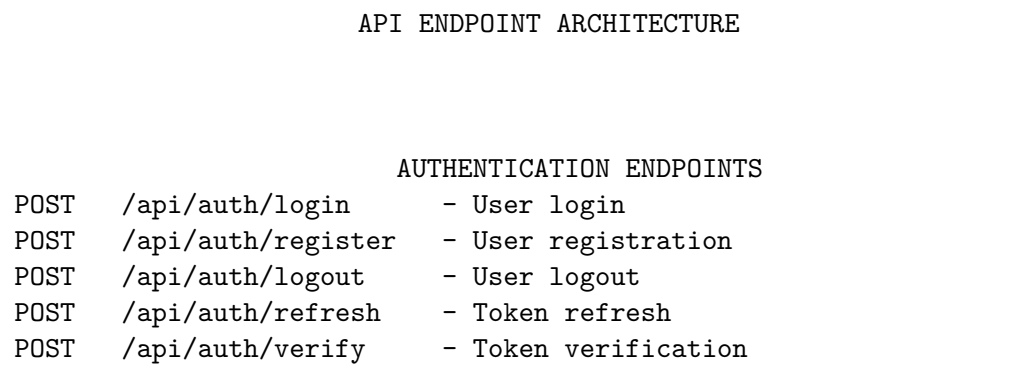
- ACID compliance
- Connection pooling
- Index optimization
- Backup and recovery

4.7.1 Database Architecture & Query Flow



4.7.2 API Endpoint Architecture Foundation

REST API with HTTP methods:



```
                              TASK ENDPOINTS
    GET    /api/tasks           - List all tasks
    POST   /api/tasks           - Create new task
    GET    /api/tasks/:id       - Get task by ID
    PUT    /api/tasks/:id       - Update task
    DELETE /api/tasks/:id       - Delete task
    PATCH  /api/tasks/:id/status- Update task status



                              USER ENDPOINTS
    GET    /api/users           - List all users
    POST   /api/users           - Create new user
    GET    /api/users/:id       - Get user by ID
    PUT    /api/users/:id       - Update user
    DELETE /api/users/:id       - Delete user
    PATCH  /api/users/:id/role  - Update user role



                              FILE ENDPOINTS
    POST   /api/files/upload    - Upload file
    GET    /api/files/:id       - Download file
    DELETE /api/files/:id       - Delete file
    GET    /api/files/task/:id  - List files for task
```

4.7.3 Server Implementation

```javascript
// server/server.js
const app = express();
const PORT = process.env.PORT || 3000;
// Middleware
app.use(helmet());
app.use(cors({
  origin: process.env.CLIENT_URL,
  credentials: true
}));
app.use(compression());
app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ extended: true }));
app.use(cookieParser());
app.use(rateLimit);
```

```
// Routes
app.use('/api/auth', authRoutes);
app.use('/api/tasks', taskRoutes);
app.use('/api/users', userRoutes);

// Error handling
app.use(errorHandler);

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

4.7.4 Authentication Middleware

```
// server/middleware/auth.js
const jwt = require('jsonwebtoken');

const authenticateToken = async (req, res, next) => {
  try {
    const authHeader = req.headers['authorization'];
    const token = authHeader && authHeader.split(' ')[1];

    if (!token) {
      return res.status(401).json({ message: 'Access token required' });
    }

    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(403).json({ message: 'Invalid or expired token' });
  }
};

const requireRole = (roles) => {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({ message: 'Authentication required' });
    }

    if (!roles.includes(req.user.role)) {
      return res.status(403).json({ message: 'Insufficient permissions' });
    }

    next();
  };
};
```

4.8 Users Table

- **id (UUID)**: Primary key, unique identifier for each user
- **email (VARCHAR)**: Unique user email
- **password (HASHED)**: Securely stored user password
- **role (ENUM)**: User role (`ADMIN`, `USER`)
- **firstName (VARCHAR)**: User first name
- **lastName (VARCHAR)**: User last name
- **createdAt (TIMESTAMP)**: Record creation timestamp
- **updatedAt (TIMESTAMP)**: Record update timestamp
- **Relationships**:
  - 1:N → Tasks
  - 1:N → Projects

4.9 Tasks Table

- **id (UUID)**: Primary key
- **title (VARCHAR)**: Task title
- **description (TEXT)**: Task details
- **status (ENUM)**: Task status (`TODO`, `IN_PROGRESS`, `DONE`)
- **priority (ENUM)**: Task priority (`LOW`, `MEDIUM`, `HIGH`, `URGENT`)
- **dueDate (TIMESTAMP)**: Optional due date
- **userId (UUID)**: Assignee (foreign key to Users)
- **projectId (UUID)**: Associated project (foreign key to Projects)
- **Relationships**:
  - 1:N → Attachments

4.10 Attachments Table

- **id (UUID)**: Primary key
- **filename (VARCHAR)**: Stored filename
- **filePath (VARCHAR)**: Path or URL to the file
- **fileSize (BIGINT)**: File size in bytes
- **mimeType (VARCHAR)**: File type
- **taskId (UUID)**: Associated task (foreign key to Tasks)
- **uploadedBy (UUID)**: Uploader user ID
- **createdAt (TIMESTAMP)**: Upload timestamp

4.11 Projects Table

- **id (UUID)**: Primary key
- **name (VARCHAR)**: Project name
- **description (TEXT)**: Optional project description
- **status (ENUM)**: Project status (`ACTIVE`, `ARCHIVED`, `COMPLETED`)
- **createdAt (TIMESTAMP)**: Creation timestamp
- **updatedAt (TIMESTAMP)**: Update timestamp
- **ownerId (UUID)**: Project owner (foreign key to Users)
- **team (ARRAY)**: List of team member IDs (foreign keys to Users)
- **Relationships**:
  - 1:N → Tasks

4.12 Database Indexing Strategy

The indexing strategy ensures fast and efficient queries:

QUERY OPTIMIZATION STRATEGY

PRIMARY INDEXES

- users.id (PRIMARY KEY)
- tasks.id (PRIMARY KEY)
- attachments.id (PRIMARY KEY)
- projects.id (PRIMARY KEY)

PERFORMANCE INDEXES

Helmet Security	HSTS Headers	CSP Headers	X-Frame Options
CORS Policy	Rate Limiting	Auth JWT	Validation Middleware

DATA VALIDATION

INPUT PARSING	SANITIZE HTML ESCAPING	VALIDATE SCHEMA CHECKING	TRANSFORM DATA FORMAT
REGEX PATTERNS	TYPE CHECKING	XSS PREVENTION	DATABASE SANITIZE

4.12.1 Prisma Client Configuration

```
// server/models/index.js
const { PrismaClient } = require('@prisma/client');

const prisma = globalThis.__prisma || new PrismaClient({
  log: process.env.NODE_ENV === 'development' ? ['query', 'error', 'warn'] : ['error'],
  errorFormat: 'pretty',
});
```

4.12.2 Task Controller Implementation

```
// server/controllers/taskController.js
const prisma = require('../models');
const { validateTaskInput } = require('../utils/validators');

const createTask = async (req, res) => {
  try {
    const { title, description, priority, dueDate, projectId } = req.body;
    const userId = req.user.id;

    // Validate input
    const validation = validateTaskInput(req.body);
    if (!validation.isValid) {
      return res.status(400).json({
        success: false,
        message: 'Validation failed',
        errors: validation.errors
      });
    }

    // Create task
    const task = await prisma.task.create({
      data: {
        title,
        description,
        priority: priority || 'MEDIUM',
        dueDate: dueDate ? new Date(dueDate) : null,
        userId,
        projectId: projectId || null
      },
      include: {
        user: {
          select: {
            id: true,
            firstName: true,
            lastName: true,
            email: true
          }
        },
        project: {
```

```
          select: {
            id: true,
            name: true
          }
        }
      }
    });

    res.status(201).json({
      success: true,
      data: task
    });

  } catch (error) {
    console.error('Task creation error:', error);
    res.status(500).json({
      success: false,
      message: 'Failed to create task'
    });
  }
};
```

4.12.3 Route Implementation

```
// server/routes/tasks.js

const router = Router();

// Apply authentication to all task routes
router.use(authenticateToken);

// Task CRUD operations
router.post('/', validateTaskInput, taskController.createTask);
router.get('/', taskController.getTasks);
router.get('/:id', taskController.getTaskById);
router.put('/:id', validateTaskInput, taskController.updateTask);
router.delete('/:id', taskController.deleteTask);

// Task status updates
router.patch('/:id/status', taskController.updateTaskStatus);

// Export the router
```

This chapter is a foundation for backend architecture with Express, Prisma, and PostgreSQL.

Chapter 5

Chapter 4: Authentication & Security

5.1 Enterprise Security Architecture

5.1.1 Multi-Layer Security Implementation

The SYNC application implements a comprehensive security architecture with multiple layers of protection:

SECURITY ARCHITECTURE

CLIENT LAYER

- Input validation and sanitization
- XSS prevention
- CSRF token management
- Secure cookie handling

TRANSPORT LAYER

- HTTPS/TLS 1.3 encryption
- Certificate pinning
- Secure headers (HSTS, CSP)
- Rate limiting and DDoS protection

APPLICATION LAYER

- JWT authentication
- Role-based access control (RBAC)
- Input validation and sanitization
- SQL injection prevention

DATA LAYER

- Database encryption at rest
- Secure connection strings
- Audit logging
- Backup encryption

5.1.2 Authentication Service Implementation

```
// server/services/authService.js
class AuthService {
  static async register(userData) {
    const { email, password, firstName, lastName } = userData;
    // Check if user already exists
    const existingUser = await prisma.user.findUnique({
      where: { email }
    });
    if (existingUser) {
      throw new Error('User already exists');
    }
    // Hash password
    const saltRounds = 12;
    const hashedPassword = await bcrypt.hash(password, saltRounds);
    // Create user
    const user = await prisma.user.create({
      data: {
        email,
        password: hashedPassword,
        firstName,
        lastName,
        role: 'USER'
      },
      select: {
        id: true,
        email: true,
        firstName: true,
        lastName: true,
        role: true,
        createdAt: true
      }
    });
    // Generate tokens
    const accessToken = jwt.sign(
      { userId: user.id, email: user.email, role: user.role },
      process.env.JWT_SECRET,
```

```
      { expiresIn: '15m' }
    );
    const refreshToken = await createRefreshToken(user.id);
    return {
      user,
      accessToken,
      refreshToken
    };
  }

  static async login(email, password) {
    // Find user
    const user = await prisma.user.findUnique({
      where: { email }
    });
    if (!user) {
      throw new Error('Invalid credentials');
    }
    // Verify password
    const isValidPassword = await bcrypt.compare(password, user.password);
    if (!isValidPassword) {
      throw new Error('Invalid credentials');
    }
    // Generate tokens
    const accessToken = jwt.sign(
      { userId: user.id, email: user.email, role: user.role },
      process.env.JWT_SECRET,
      { expiresIn: '15m' }
    );
    const refreshToken = await createRefreshToken(user.id);
    return {
      user: {
        id: user.id,
        email: user.email,
        firstName: user.firstName,
        lastName: user.lastName,
        role: user.role
      },
      accessToken,
      refreshToken
    };
  }

  static async refreshAccessToken(refreshToken) {
    const payload = await verifyRefreshToken(refreshToken);
    const user = await prisma.user.findUnique({
      where: { id: payload.userId },
      select: {
        id: true,
        email: true,
        role: true
```

```
      }
    });
    if (!user) {
      throw new Error('User not found');
    }
    const newAccessToken = jwt.sign(
      { userId: user.id, email: user.email, role: user.role },
      process.env.JWT_SECRET,
      { expiresIn: '15m' }
    );
    return { accessToken: newAccessToken };
  }
}
```

5.1.3 CSRF Protection Middleware

```
// server/middleware/csrf.js
const crypto = require('crypto');

const generateCSRFToken = (req, res, next) => {
  try {
    // Generate CSRF token
    const csrfToken = crypto.randomBytes(32).toString('hex');
    // Store token in session or memory
    req.session = req.session || {};
    req.session.csrfToken = csrfToken;
    res.locals.csrfToken = csrfToken;
    next();
  } catch (error) {
    next(error);
  }
};

const validateCSRFToken = (req, res, next) => {
  try {
    const { csrfToken } = req.body;
    const sessionToken = req.session?.csrfToken;
    if (!csrfToken || !sessionToken || csrfToken !== sessionToken) {
      return res.status(403).json({
        success: false,
        message: 'CSRF token validation failed'
      });
    }
    // Clear used token
    delete req.session.csrfToken;
    next();
  } catch (error) {
    next(error);
  }
};
```

This chapter provides exmaples of security implementation for applications with multi-layer protection, authentication, and authorization systems.

Chapter 6

Chapter 5: Frontend Architecture

6.1 React Application Structure

6.1.1 Main Application Component

```jsx
// client/src/App.jsx


const App = () => {
  return (
    <AuthProvider>
      <Router>
        <div className="min-h-screen bg-gray-50">
          <Routes>
            {/* Public routes */}
            <Route path="/login" element={<Login />} />
            <Route path="/register" element={<Register />} />
            {/* Protected routes */}
            <Route path="/" element={<ProtectedRoute><Layout /></ProtectedRoute>}>
              <Route index element={<Dashboard />} />
              <Route path="tasks" element={<Tasks />} />
              <Route path="projects" element={<Projects />} />
              <Route path="users" element={<Users />} />
            </Route>
          </Routes>
          <Toaster
            position="top-right"
            toastOptions={{
              duration: 4000,
              style: {
                background: '#363636',
                color: '#fff',
              },
            }}
          />
        </div>
      </Router>
```

```
    </AuthProvider>
  );
};
```

6.2 Design System & UI Architecture

Our design system leverages TailwindCSS for consistent, responsive, and beautiful interfaces:

6.2.1 Component Architecture

The frontend follows a modular component architecture with:

- **Atomic Design Principles**: Building blocks from atoms to organisms
- **Reusable Components**: Shared UI elements across the application
- **Responsive Design**: Mobile-first approach with adaptive breakpoints
- **Accessibility**: WCAG 2.1 AA compliance with screen reader support
- **Theme System**: Light/dark mode with custom branding support
- **Performance**: Code splitting and lazy loading for optimal performance

6.2.2 State Management with Zustand

Zustand provides lightweight, scalable state management:

- **Simple API**: Minimal boilerplate compared to Redux
- **Type Safety**: Full JavaScript support with type definitions
- **Middleware Support**: DevTools, persistence, and custom middleware
- **React Integration**: Hooks-based API for seamless React integration

```
// client/src/stores/taskStore.js


const useTaskStore = create(
  devtools(
    persist(
      (set, get) => ({
        // State
        tasks: [],
        loading: false,
        error: null,
        filters: {
          status: null,
          priority: null,
          projectId: null,
          search: ''
        },
        pagination: {
          page: 1,
          limit: 10,
          total: 0
        },

        // Actions
```

```
  setTasks: (tasks) => set({ tasks }),
  setLoading: (loading) => set({ loading }),
  setError: (error) => set({ error }),
  setFilters: (filters) => set({ filters }),
  setPagination: (pagination) => set({ pagination }),
  // Async actions
  fetchTasks: async (params = {}) => {
    set({ loading: true, error: null });
    try {
      const response = await taskService.getTasks(params);
      set({
        tasks: response.data,
        pagination: response.pagination,
        loading: false
      });
    } catch (error) {
      set({
        error: error.message,
        loading: false
      });
    }
  },

  createTask: async (taskData) => {
    set({ loading: true, error: null });
    try {
      const newTask = await taskService.createTask(taskData);
      set(state => ({
        tasks: [newTask, ...state.tasks],
        loading: false
      }));
      return newTask;
    } catch (error) {
      set({
        error: error.message,
        loading: false
      });
      throw error;
    }
  },


  getFilteredTasks: () => {
    const state = get();
    let filtered = state.tasks;

    if (state.filters.status) {
      filtered = filtered.filter(task => task.status === state.filters.status);
    }

    if (state.filters.priority) {
```

```
            filtered = filtered.filter(task => task.priority === state.filters.priority);
          }

          if (state.filters.projectId) {
            filtered = filtered.filter(task => task.projectId === state.filters.projectId);
          }

          if (state.filters.search) {
            const searchLower = state.filters.search.toLowerCase();
            filtered = filtered.filter(task =>
              task.title.toLowerCase().includes(searchLower) ||
              task.description?.toLowerCase().includes(searchLower)
            );
          }

          return filtered;
        }
      }),
      {
        name: 'task-store',
        partialize: (state) => ({
          filters: state.filters,
          pagination: state.pagination
        })
      }
    ),
    {
      name: 'task-store'
    }
  )
);
```

6.2.3 Form Components

```
const TaskForm = ({ task, onSubmit, onCancel, mode = 'create' }) => {
  const { createTask, updateTask, loading } = useTaskStore();
  const { projects } = useProjectStore();
  const initialValues = {
    title: task?.title || '',
    description: task?.description || '',
    priority: task?.priority || 'MEDIUM',
    status: task?.status || 'TODO',
    dueDate: task?.dueDate ? new Date(task.dueDate).toISOString().split('T')[0] : '',
    projectId: task?.projectId || ''
  };

  const validationSchema = Yup.object({
    title: Yup.string()
      .min(3, 'Title must be at least 3 characters')
      .max(200, 'Title must not exceed 200 characters')
```

```
    .required('Title is required'),
});

const handleSubmit = async (values, { setSubmitting, resetForm }) => {
  try {
    if (mode === 'create') {
      await createTask(values);
    } else {
      await updateTask(task.id, values);
    }

    onSubmit?.(values);
    resetForm();
  } catch (error) {
    console.error('Task submission error:', error);
  } finally {
    setSubmitting(false);
  }
};

return (
  <div className="bg-white rounded-lg shadow-sm border p-6">
    <h2 className="text-xl font-semibold text-gray-900 mb-6">
      {mode === 'create' ? 'Create New Task' : 'Edit Task'}
    </h2>
    <Formik
      initialValues={initialValues}
      validationSchema={validationSchema}
      onSubmit={handleSubmit}
    >
      {({ isSubmitting, isValid, dirty }) => (
        <Form className="space-y-6">
          <div>
            <label htmlFor="title" className="block text-sm font-medium text-gray-700 mb-2
              Title *
            </label>
            <Field
              type="text"
              id="title"
              name="title"
              className="w-full px-3 py-2 border border-gray-300 rounded-md shadow-sm focu
              placeholder="Enter task title"
            />
            <ErrorMessage name="title" component="div" className="mt-1 text-sm text-red-60
          </div>
          <div className="flex justify-end space-x-3 pt-4">
            <Button
              type="button"
              variant="outline"
              onClick={onCancel}
              disabled={isSubmitting}
```

```
              >
                Cancel
              </Button>
              <Button
                type="submit"
                disabled={isSubmitting || !isValid || !dirty}
                loading={isSubmitting}
              >
                {mode === 'create' ? 'Create Task' : 'Update Task'}
              </Button>
            </div>
          </Form>
        )}
      </Formik>
    </div>
  );
};
```

6.2.4 Custom Hooks

```
function useApi(options = {}) {
  const {
    url,
    method = 'GET',
    body = null,
    headers = {},
    immediate = false,
    onSuccess,
    onError
  } = options;

  const execute = useCallback(async (customOptions = {}) => {
    const finalOptions = { ...options, ...customOptions };
    const {
      url: finalUrl,
      method: finalMethod,
      body: finalBody,
      headers: finalHeaders
    } = finalOptions;
    if (!finalUrl) return;
    setLoading(true);
    setError(null);
    try {
      const requestHeaders = {
        'Content-Type': 'application/json',
        ...finalHeaders
      };
      if (accessToken) {
        requestHeaders.Authorization = `Bearer ${accessToken}`;
      }
```

```
      const response = await fetch(finalUrl, {
        method: finalMethod,
        headers: requestHeaders,
        body: finalBody ? JSON.stringify(finalBody) : null
      });
      if (!response.ok) {
        if (response.status === 401) {
          logout();
          throw new Error('Authentication required');
        }
        throw new Error(`HTTP error! status: ${response.status}`);
      }
      const responseData = await response.json();
      setData(responseData);
      onSuccess?.(responseData);
      return responseData;
    } catch (err) {
      const errorMessage = err.message || 'An error occurred';
      setError(errorMessage);
      onError?.(err);
      throw err;
    } finally {
      setLoading(false);
    }
  }, [url, method, body, headers, accessToken, logout, onSuccess, onError]);

  useEffect(() => {
    if (immediate && url) {
      execute();
    }
  }, [immediate, url, execute]);

  return {
    data,
    loading,
    error,
    execute,
    setData,
    setError
  };
}
```

6.2.5 Kanban Board Components

```
const KanbanCard = React.memo(({ task, index, onEdit, onDelete }) => {
  const timeAgo = useMemo(() => {
    if (!task.updatedAt) return '';
    return formatDistanceToNow(new Date(task.updatedAt), { addSuffix: true });
  }, [task.updatedAt]);
```

```
  const handleEdit = useCallback(() => {
    onEdit?.(task);
  }, [task, onEdit]);

  const handleDelete = useCallback(() => {
    if (window.confirm('Are you sure you want to delete this task?')) {
      onDelete?.(task.id);
    }
  }, [task.id, onDelete]);

  return (
    <Draggable draggableId={task.id} index={index}>
      {(provided, snapshot) => (
        <div
          ref={provided.innerRef}
          {...provided.draggableProps}
          {...provided.dragHandleProps}
          className={`
            bg-white rounded-lg shadow-sm border p-4 mb-3 cursor-move
            ${snapshot.isDragging ? 'shadow-lg rotate-2' : ''}
            hover:shadow-md transition-all duration-200
          `}
        >
          <div className="flex items-start justify-between mb-3">
            <h3 className="font-medium text-gray-900 text-sm leading-tight line-clamp-2">
              {task.title}
            </h3>
            <div className="flex items-center space-x-2 ml-2">
              <PriorityBadge priority={task.priority} />
              <StatusBadge status={task.status} />
            </div>
          </div>

          {task.description && (
            <p className="text-gray-600 text-xs mb-3 line-clamp-3">
              {task.description}
            </p>
          )}
          </div>
        </div>
      )}
    </Draggable>
  );
});

KanbanCard.displayName = 'KanbanCard';
```

6.3 Performance Optimization

6.3.1 Code Splitting and Lazy Loading

- **Route-based Code Splitting**: Each page is loaded only when needed
- **Component Lazy Loading**: Heavy components are loaded on demand
- **Bundle Optimization**: Tree shaking and dead code elimination
- **Image Optimization**: WebP format with fallbacks and lazy loading
- **Caching Strategies**: Service worker for offline support

6.3.2 Responsive Design Implementation

- **Mobile-First Approach**: Design starts with mobile and scales up
- **Flexible Grid System**: CSS Grid and Flexbox for adaptive layouts
- **Breakpoint Management**: Consistent breakpoints across components
- **Touch-Friendly Interface**: Optimized for mobile interactions
- **Performance Monitoring**: Real-time performance metrics

This chapter offer a modern frontend architecture with React, Zustand state management, and component design for scalable applications.

Chapter 7

Chapter 6: Task Management & Kanban

7.1 Kanban Board & Task Management Diagrams

7.1.1 Kanban Board Architecture Overview

KANBAN BOARD ARCHITECTURE

TODO COLUMN

- Task 1
- Task 2
- Task 3

IN PROGRESS

- Task 4
- Task 5

REVIEW

- Task 6
- Task 7

DRAG & DROP SYSTEM

- Visual
- Intuitive
- Responsive

STATUS UPDATES

- Real-time
- Persistence
- Notifications

WORKFLOW RULES

- Validation
- Permissions
- Automation

7.1.2 Task Status Flow & Workflow Management

TASK WORKFLOW DIAGRAM

DRAFT

- Created
- Not Assigned

PENDING

- Approved
- Assigned
- Scheduled

ACTIVE

- In Progress
- Updated
- Tracked

REVIEW

- Completed
- Ready for Approval

ARCHIVED

- Historical
- Reference
- Analytics

BLOCKED

- Dependencies
- Issues
- Resolution

ON HOLD

- Waiting
- Resources
- Approval

COMPLETED

- Delivered
- Closed
- Archive

7.1.3 Task Assignment & User Management

TASK ASSIGNMENT SYSTEM

TASK CREATION

- Title
- Description
- Priority

USER SELECTION

- Skills
- Availability
- Workload

ASSIGNMENT PROCESS

- Notification
- Permission
- Tracking

VALIDATION & APPROVAL

- Business Rules
- Permissions

WORKFLOW INTEGRATION

- Status Updates
- Notifications

MONITORING & REPORTING

- Progress
- Performance
- Analytics

7.1.4 Priority Management & Sorting Algorithms

PRIORITY MANAGEMENT SYSTEM

PRIORITY LEVELS

URGENT	HIGH	MEDIUM	LOW
• Red	• Orange	• Yellow	• Green
• Top	• High	• Normal	• Low
• Immediate	• Important	• Standard	• Optional

SORTING ALGORITHMS

- Priority-based sorting (High to Low)
- Due date sorting (Earliest first)
- Creation date sorting (Newest first)
- Assignee-based grouping
- Project-based organization
- Custom sorting rules

Priority Management Features: - **Visual Indicators**: Color-coded priority levels for quick identification - **Smart Sorting**: Automatic sorting based on priority, due date, and creation time - **Workload Balancing**: Intelligent task distribution across team members - **Deadline Management**: Automatic notifications for approaching due dates - **Escalation Rules**: Automatic priority escalation for overdue tasks - **Capacity Planning**: Workload visualization and resource allocation

7.1.5 Real-time Collaboration & Updates

REAL-TIME COLLABORATION

USER ACTION	WEBSOCKET CONNECTION	SERVER PROCESSING	BROADCAST TO TEAM
TASK UPDATE	STATE CHANGE	VALIDATE & PROCESS	UPDATE CLIENTS

Real-time Features: - **Live Updates**: Instant synchronization across all team members - **Conflict Resolution**: Automatic conflict detection and resolution - **Offline Support**: Local caching with sync when online - **Activity Feed**: Real-time activity tracking and notifications - **Collaborative Editing**: Multiple users can work simultaneously - **Version History**: Com-

plete audit trail of all changes

```
const useKanbanStore = create(
  devtools(
    (set, get) => ({

const initialColumns = {
  TODO: [],
  IN_PROGRESS: [],
  REVIEW: [],
  DONE: [],
};

      columns: initialColumns,
      isDragging: false,
      draggedTask: null,
      targetColumn: null,
      initializeBoard: (tasks) => {
        const columns = { ...initialColumns };

        tasks.forEach((task) => {
          if (columns[task.status]) {
            columns[task.status].push(task);
          }
        });
        // Sort tasks by priority and creation date
        Object.keys(columns).forEach((status) => {
          columns[status].sort((a, b) => {
            const priorityOrder = { URGENT: 4, HIGH: 3, MEDIUM: 2, LOW: 1 };
            const priorityDiff = priorityOrder[b.priority] - priorityOrder[a.priority];

            if (priorityDiff !== 0) return priorityDiff;

            return new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime();
          });
        });
        set({ columns });
      },

      moveTask: async (taskId, fromStatus, toStatus) => {
        try {
          // Update task status via API
          const updatedTask = await taskService.updateTaskStatus(taskId, toStatus);
          // Update local state
          set((state) => {
            const newColumns = { ...state.columns };
            // Remove from source column
            newColumns[fromStatus] = newColumns[fromStatus].filter(
              (task) => task.id !== taskId
            );
            // Add to target column
            newColumns[toStatus] = [...newColumns[toStatus], updatedTask];
```

```
            // Sort target column
            newColumns[toStatus].sort((a, b) => {
              const priorityOrder = { URGENT: 4, HIGH: 3, MEDIUM: 2, LOW: 1 };
              const priorityDiff = priorityOrder[b.priority] - priorityOrder[a.priority];
              if (priorityDiff !== 0) return priorityDiff;

              return new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime();
            });

            return { columns: newColumns };
          });
          return updatedTask;
        } catch (error) {
          console.error('Failed to move task:', error);
          throw error;
        }
      },
    })
    {
      name: 'kanban-store',
    }
  )
);
```

This chapter - task management with Kanban boards and role-based access control.

Chapter 8

Chapter 7: File Handling & Storage

8.1 Overview

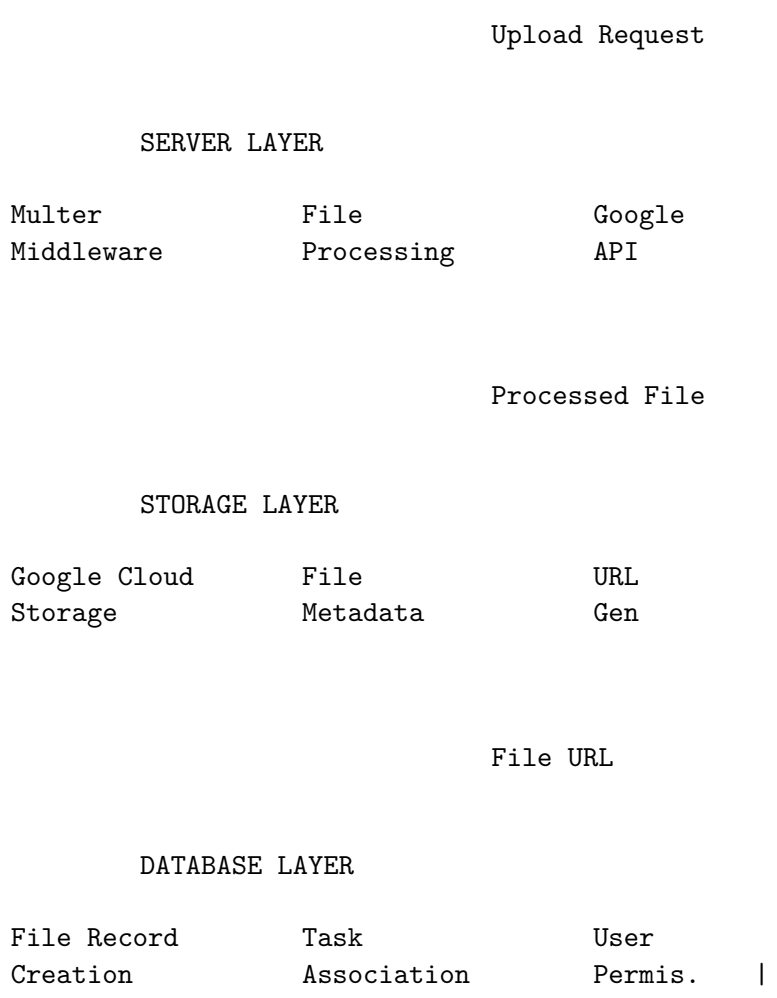
File handling and storage represent critical components of any production application. In Sync, i implement a solution that combines security, performance, and scalability through Google Cloud Storage integration. This chapter explores the technical implementation, security considerations, and real-world deployment strategies.

8.2 File Management Architecture

8.2.1 System Overview

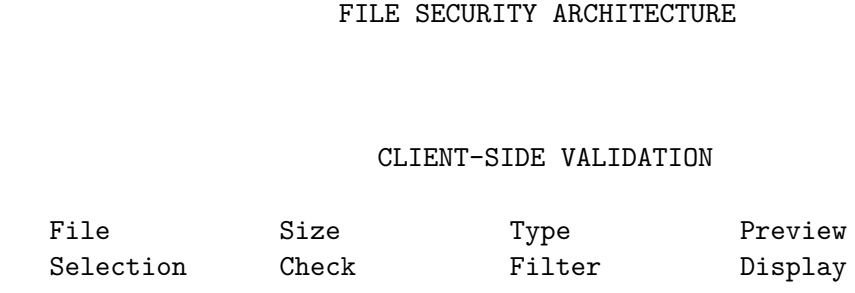
File Upload Flow Architecture:

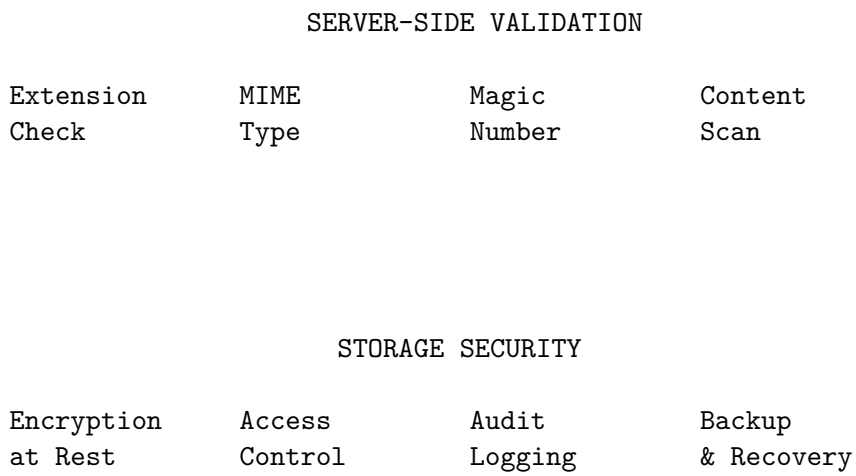
```
                CLIENT LAYER

    File Input          Drag & Drop         Paste
    Component           Interface           Support


                                    File Selection


          VALIDATION LAYER

    File Type           File Size           Content
    Validation          Validation          Check


                                    Validated File


          UPLOAD LAYER

    Multipart           Progress            Error
    Form Data           Tracking            Handling
```



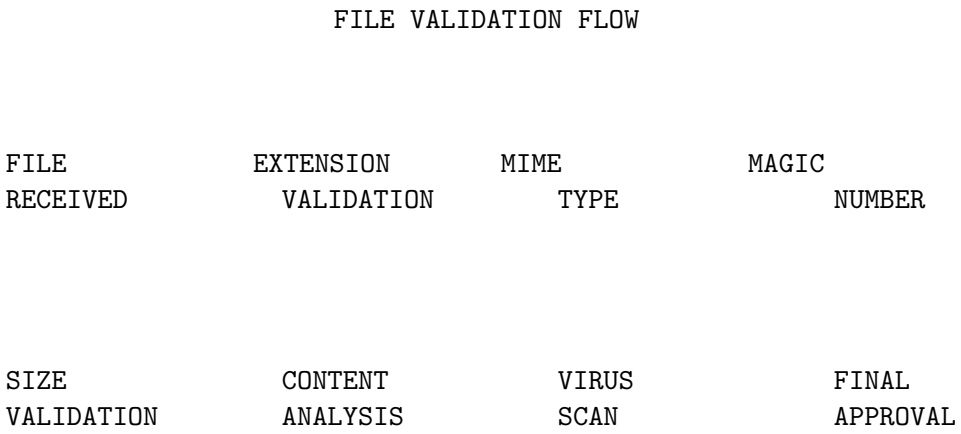
8.3 Advanced Security & Validation Diagrams

8.3.1 Multi-Layer Security Architecture





8.3.2 File Validation & Sanitization Flow



Security Features: - **Multi-Layer Validation**: File extension, MIME type, and magic number verification - **Virus Scanning**: Integration with security services for malware detection - **Content Analysis**: Deep inspection of file contents for threats - **Access Control**: Role-based permissions and audit logging - **Encryption**: AES-256 encryption for all stored files

8.3.3 Content Security

Virus Scanning Integration

```
// Integration with ClamAV or similar
const scanFile = async (fileBuffer) => {
    try {
        const result = await virusScanner.scan(fileBuffer);
        if (result.isInfected) {
            throw new Error('File contains malware');
```

```
        }
        return true;
    } catch (error) {
        console.error('Virus scan failed:', error);
        // In production, reject files if scan fails
        throw new Error('Security scan failed');
    }
};
```

Image Processing and Sanitization

```
const processImage = async (fileBuffer, mimeType) => {
    const sharp = require('sharp');

    try {
        // Remove EXIF data (potential privacy risk)
        const processedImage = await sharp(fileBuffer)
            .removeExif()
            .resize(1920, 1080, { fit: 'inside' }) // Max dimensions
            .jpeg({ quality: 85 }) // Optimize quality
            .toBuffer();

        return processedImage;
    } catch (error) {
        throw new Error('Image processing failed');
    }
};
```

This chapter demonstrates some example from my app how to implement file management that balances security, performance. The Google Cloud Storage integration provides a robust foundation for handling file uploads in production environments.

Chapter 9

Chapter 8: Deployment & Scaling

9.1 Overview

Deployment and scaling are critical phases in the application lifecycle. This chapter covers production deployment strategies, containerization with Docker, and both vertical and horizontal scaling approaches. We'll explore how to take Sync from development to production and prepare it for enterprise-scale usage.

9.2 Containerization with Docker

9.2.1 Development Dockerfile

```
# Dockerfile.dev
FROM node:18-alpine

# Set working directory
WORKDIR /app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production

ENTRYPOINT ["dumb-init", "--"]
CMD ["node", "src/server.js"]
```

9.2.2 Docker Compose for Production

```
# docker-compose.prod.yml
version: '3.8'

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile.prod
```

```
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
      - DATABASE_URL=${DATABASE_URL}
      - JWT_SECRET=${JWT_SECRET}
      - GOOGLE_CLOUD_PROJECT_ID=${GOOGLE_CLOUD_PROJECT_ID}
    depends_on:
      - db
      - redis
    restart: unless-stopped
    deploy:
      replicas: 3
      resources:
        limits:
          cpus: '1.0'
          memory: 1G
        reservations:
          cpus: '0.5'
          memory: 512M

  db:
    image: postgres:15-alpine
    environment:
      - POSTGRES_DB=${POSTGRES_DB}
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
    volumes:
      - postgres_prod_data:/var/lib/postgresql/data
    restart: unless-stopped
    deploy:
      resources:
        limits:
          cpus: '2.0'
          memory: 4G

  redis:
    image: redis:7-alpine
    command: redis-server --appendonly yes --requirepass ${REDIS_PASSWORD}
    volumes:
      - redis_prod_data:/data
    restart: unless-stopped
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 512M

  nginx:
    image: nginx:alpine
    ports:
```

```
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
      - ./nginx/ssl:/etc/nginx/ssl
    depends_on:
      - app
    restart: unless-stopped

volumes:
  postgres_prod_data:
  redis_prod_data:
```

9.3 Scaling Strategies

9.3.1 Vertical Scaling

Server Resource Optimization

```
// server.js - Optimized for vertical scaling
const cluster = require('cluster');
const os = require('os');

if (cluster.isMaster) {
    const numCPUs = os.cpus().length;

    console.log(`Master ${process.pid} is running`);
    console.log(`Forking for ${numCPUs} CPUs`);

    // Fork workers
    for (let i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    cluster.on('exit', (worker, code, signal) => {
        console.log(`Worker ${worker.process.pid} died`);
        // Replace the dead worker
        cluster.fork();
    });
} else {
    // Worker process
    require('./src/app');
    console.log(`Worker ${process.pid} started`);
}
```

9.3.2 Horizontal Scaling

Load Balancer Configuration

```
// loadBalancer.js
const http = require('http');
const httpProxy = require('http-proxy');
```

```
const proxy = httpProxy.createProxyServer({});

const servers = [
    'http://app1:3000',
    'http://app2:3000',
    'http://app3:3000'
];

let currentServer = 0;

const server = http.createServer((req, res) => {
    // Round-robin load balancing
    const target = servers[currentServer];
    currentServer = (currentServer + 1) % servers.length;

    proxy.web(req, res, { target });
});

server.listen(8080);
```

This chapter demonstrates how to take the Task Manager App from development to production and scale it to handle workloads. The combination of containerization and load balancing a robust foundation for growth. ## Bibliography # Bibliography

9.4 Primary Technologies & Frameworks

9.4.1 Backend Technologies

Node.js - Node.js Foundation. (2024). Node.js Documentation. https://nodejs.org/docs/

Express.js - StrongLoop, IBM, and other contributors. (2024). Express.js - Fast, unopinionated, minimalist web framework for Node.js. https://expressjs.com/

PostgreSQL - PostgreSQL Global Development Group. (2024). PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/

Prisma ORM - Prisma Documentation. (2024). Prisma Client, Prisma Migrate, and Prisma Studio. https://www.prisma.io/docs/

9.4.2 Frontend Technologies

React - Facebook, Inc. (2024). React – A JavaScript library for building user interfaces. https://reactjs.org/

Vite - Vite Documentation. (2024). Build tool that aims to provide a faster and leaner development experience. https://vitejs.dev/guide/

JavaScript - ECMAScript. (2024). ECMAScript Language Specification. https://tc39.es/ecma262/

TailwindCSS - Tailwind CSS Documentation. (2024). Rapidly build modern websites without ever leaving your HTML. https://tailwindcss.com/docs

9.4.3 State Management

Zustand - Zustand Documentation. (2024). Simple state management for React. https://github.com/pmndrs/zustand#readme

9.4.4 Form Management

Formik - Formik Documentation. (2024). Form state management and validation for React. https://formik.org/docs/overview

Yup - Yup Documentation. (2024). JavaScript object schema validator and object parser. https://github.com/jquense/yup#api

9.4.5 Authentication & Security

JSON Web Tokens (JWT) - Auth0. (2024). JWT.io - JSON Web Token Debugger. https://jwt.io/

Bcrypt - Bcrypt Documentation. (2024). bcrypt - A library to help you hash passwords. https://github.com/dcodeIO/bcrypt.js

Helmet.js - Helmet Documentation. (2024). Security middleware for Express.js. https://helmetjs.github.io/docs/

9.4.6 File Handling & Storage

Google Cloud Storage - Google Cloud Documentation. (2024). Store and serve large amounts of data. https://cloud.google.com/storage/docs

9.4.7 Development Tools

Git - Torvalds, L., & Hamano, J. (2024). Git - Distributed version control system. https://git-scm.com/

9.4.8 Deployment & DevOps

Docker - Docker Documentation. (2024). Container platform for developers and DevOps. https://docs.docker.com/

9.4.9 Security Standards & Best Practices

CORS - MDN Web Docs. (2024). Cross-Origin Resource Sharing (CORS). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

CSRF Protection - MDN Web Docs. (2024). Cross-Site Request Forgery (CSRF). https://developer.mozilla.org/en-US/docs/Glossary/CSRF

9.4.10 Database Design & Migration

Prisma Migrations - Prisma Documentation. (2024). Database schema migrations. https://www.prisma.io/docs/concepts/components/prisma-migrate

---

Book Generation Complete!

Generated on: marți 2 septembrie 2025, 03:32:01 +0300 Total files processed: 10