Library Management System - Project Structure Documentation

Project Structure

```
app/
├── __init__.py
├── main.py
├── api/
│   └── v1/
│       ├── __init__.py
│       └── routers/
│           ├── books.py
│           ├── members.py
│           ├── loans.py
│           └── reports.py
├── config/
│   ├── __init__.py
│   ├── database.py
│   └── settings.py
├── models/
│   └── domain/
│       ├── __init__.py
│       ├── base.py
│       ├── book.py
│       ├── member.py
│       ├── loan.py
│       └── types.py
├── schemas/
│   └── v1/
│       ├── __init__.py
```

```
│       ├── requests/
│       │   ├── book.py
│       │   ├── member.py
│       │   └── loan.py
│       └── responses/
│           ├── book.py
│           ├── member.py
│           └── loan.py
├── services/
│   ├── __init__.py
│   ├── book_service.py
│   ├── member_service.py
│   ├── loan_service.py
│   └── report_service.py
└── utils/
    ├── __init__.py
    ├── constants.py
    ├── exceptions.py
    ├── helpers.py
    └── logger.py
```

Directory Structure Documentation

1. api/

Purpose: Contains all API endpoints and route handlers.

- `v1/`: API version control
- `routers/`: Route handlers for different entities
  - `books.py`: Book-related endpoints (CRUD)
  - `members.py`: Member management endpoints
  - `loans.py`: Book loan and return endpoints
  - `reports.py`: Reporting and analytics endpoints

Example Endpoints:

- `GET /books`: List all books
- `POST /books`: Add new book
- `GET /members/{id}`: Get member details
- `POST /loans`: Create new loan

2. config/

Purpose: Application configuration and settings.

- `database.py`: Database connection configuration
- `settings.py`: Application settings and environment variables

Key Components:

- Database URL and credentials
- Environment-specific settings
- API configurations
- Logging settings

3. models/

Purpose: Database models and domain entities.

- `base.py`: Base model class with common fields
- Entity-specific models:
  - `book.py`: Book model (ISBN, title, author, etc.)
  - `member.py`: Library member model
  - `loan.py`: Book loan records
  - `types.py`: Custom types and enums

Example Model:

```
class Book(Base):
    __tablename__ = "books"
```

```
    id = Column(Integer, primary_key=True)
    isbn = Column(String, unique=True)
    title = Column(String, nullable=False)
    author = Column(String, nullable=False)
    status = Column(Enum(BookStatus))
```

4. schemas/

Purpose: Request/Response data validation schemas.

- `requests/`: Input validation schemas
- `responses/`: Output formatting schemas

Benefits:

- Input validation
- Type checking
- Response standardization
- API documentation

5. services/

Purpose: Business logic implementation.

- Entity-specific services:
  - `book_service.py`: Book management logic
  - `member_service.py`: Member operations
  - `loan_service.py`: Loan processing
  - `report_service.py`: Report generation

Responsibilities:

- Data processing
- Business rule enforcement
- Transaction management
- Error handling

6. utils/

Purpose: Utility functions and helpers.

- `constants.py`: System constants and enums
- `exceptions.py`: Custom exception classes
- `helpers.py`: Helper functions
- `logger.py`: Logging configuration

Common Utilities:

- Date formatting
- Input validation
- Error handling
- Logging

Best Practices

1. Code Organization

- Keep related functionality together
- Use meaningful file names
- Maintain consistent naming conventions
- Document complex logic

2. Error Handling

- Use custom exceptions
- Implement proper error logging
- Return meaningful error messages
- Handle edge cases

3. Documentation

- Include docstrings

- Document API endpoints
- Maintain README files
- Add code comments for complex logic

4. Testing

- Unit tests for services
- Integration tests for APIs
- Test edge cases
- Maintain test coverage

Development Guidelines

1. API Development

- Use FastAPI decorators
- Implement proper validation
- Document all endpoints
- Follow REST principles

2. Database Operations

- Use SQLAlchemy ORM
- Implement proper migrations
- Handle transactions properly
- Optimize queries

3. Code Style

- Follow PEP 8
- Use type hints
- Implement proper logging
- Keep functions focused

4. Security

- Implement authentication
- Validate inputs
- Sanitize data
- Handle sensitive information properly

Getting Started

1. Clone the repository
2. Install dependencies:

   ```
   pip install -r requirements.txt
   ```

3. Set up environment variables
4. Run database migrations
5. Start the application:

   ```
   uvicorn app.main:app --reload
   ```

Deployment

1. Build the application
2. Configure environment variables
3. Run database migrations
4. Start the application server
5. Monitor logs and performance

This structure provides a solid foundation for building a library management system with clean architecture and proper separation of concerns.