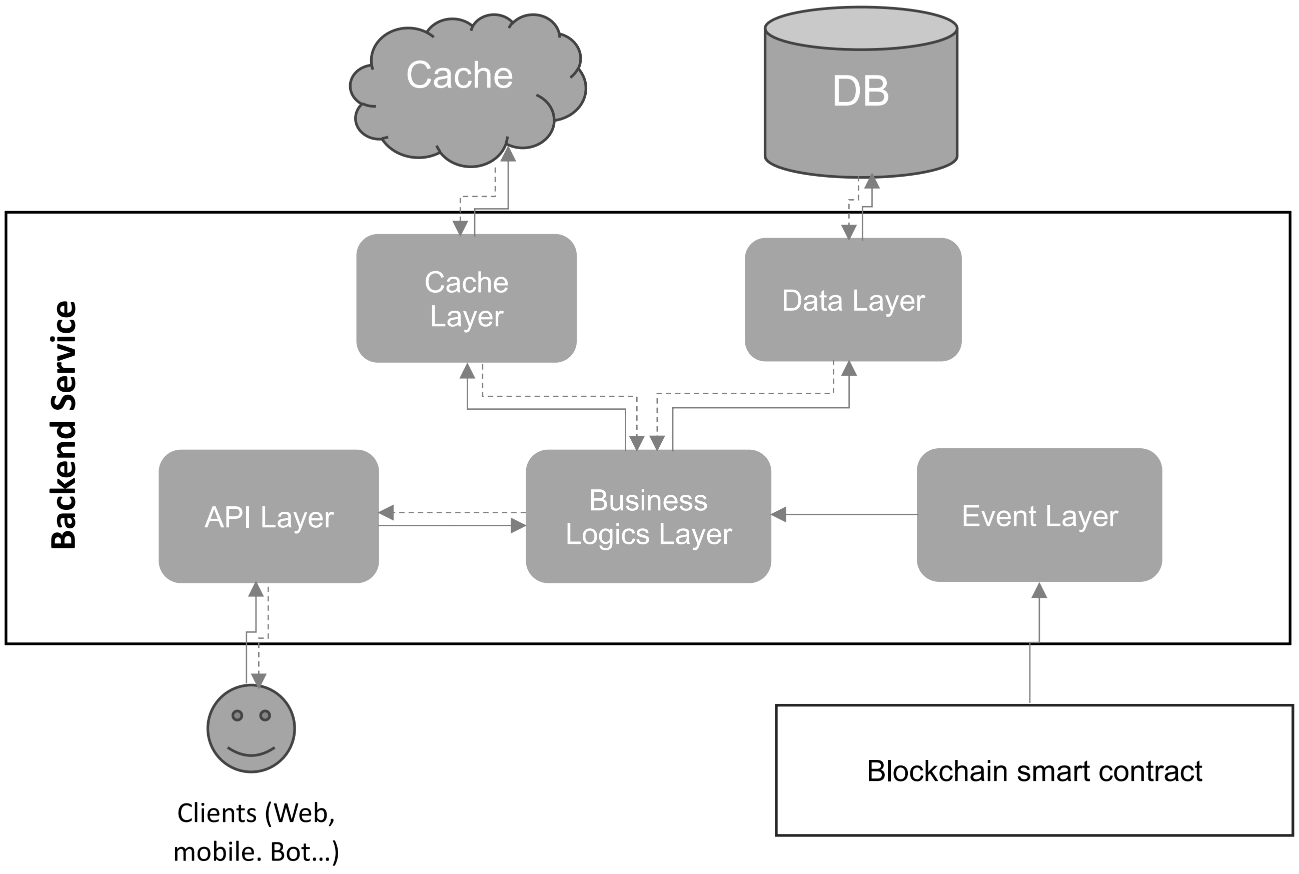
Libre Investment Platform Test Project Backend

1. Architecture Diagram



This is the architecture diagram of the investment fund backend service system.

Backend service consists of 5 components (layers).

A. Event layer

This component listens to events occurring in the FundToken smart contract. Whenever an event occurs, it catches it and passes it to the Business Logics layer. This component is the interface with smart contract in synchronizing the blockchain state and the DB state.

B. API Layer

This component listens to the requests from the clients. Whenever a request is received, it passes it to the Business Logics layer. This component is the interface with the clients.

C. Business Logics Layer

This component is the center of this system. All the data are processed and all the business logics are performed in this component.

D. Data Layer

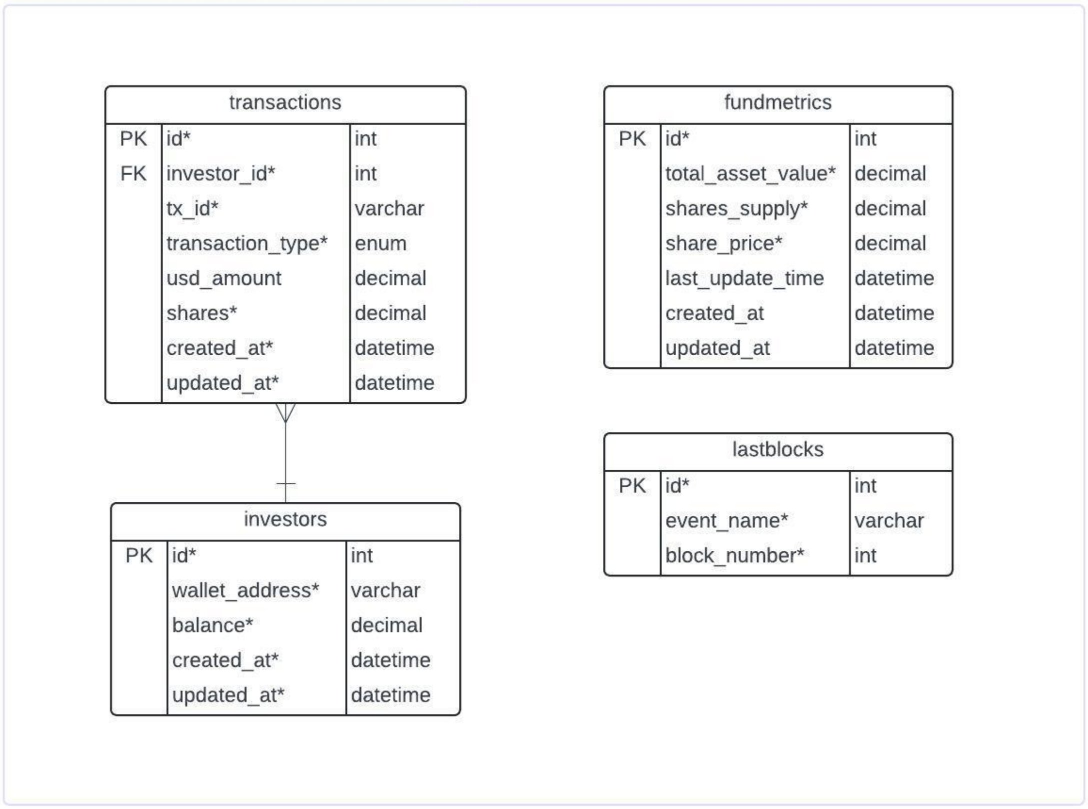
Whenever the business logic layer needs to read or write data from DB store, it calls interfaces of the Data Layer. Data layer interacts with database directly to read or write data in it.

E. Cache Layer

This component is very important in enhancing performance of overall system. Frequently called data that is not modified permanently can be stored in memory-style cache and reused later, which increases the performance. Whenever the business logic layer needs to read or write cache, it calls interfaces of this Cache Layer and Cache Layer interacts with the cache to perform caching.

2. Data Model Design

Libre Investment Platform DB Structure (MySQL)



transactions		
PK	id*	int
FK	investor_id*	int
	tx_id*	varchar
	transaction_type*	enum
	usd_amount	decimal
	shares*	decimal
	created_at*	datetime
	updated_at*	datetime

investors		
PK	id*	int
	wallet_address*	varchar
	balance*	decimal
	created_at*	datetime
	updated_at*	datetime

fundmetrics		
PK	id*	int
	total_asset_value*	decimal
	shares_supply*	decimal
	share_price*	decimal
	last_update_time	datetime
	created_at	datetime
	updated_at	datetime

lastblocks		
PK	id*	int
	event_name*	varchar
	block_number*	int

3. Implementation Details

A. Blockchain interactions

Investment and redemption should be done in the frontend using wallets. It is mandatory for security and privacy. This indicates that there is no need to have an Investment and Redemption endpoint in the backend.

If the user performs investment or redemption, the frontend can acknowledge it to the backend. Or the backend can get to know Investment and redemption by listening to events of the smart contract.

There are 2 possible approaches to implement blockchain interactions.

- User performs investment or redemption in the frontend and the frontend informs it to the backend via some RESTful API endpoints.
- User performs investment or redemption in the frontend and the frontend doesn't inform this to the backend. But the backend should listen to events of the smart contract.

I chose the second approach after answering questions about this to the tech team.

I thought this approach is more reliable because events can be acquired later even though the server is missing some of them because of the backend server downtime.

In the second option, there should be an important consideration about backend downtime for fully synchronizing the DB with the blockchain state.

The backend should listen to events not only for real-time events, but also for past events that can be missing in the DB.

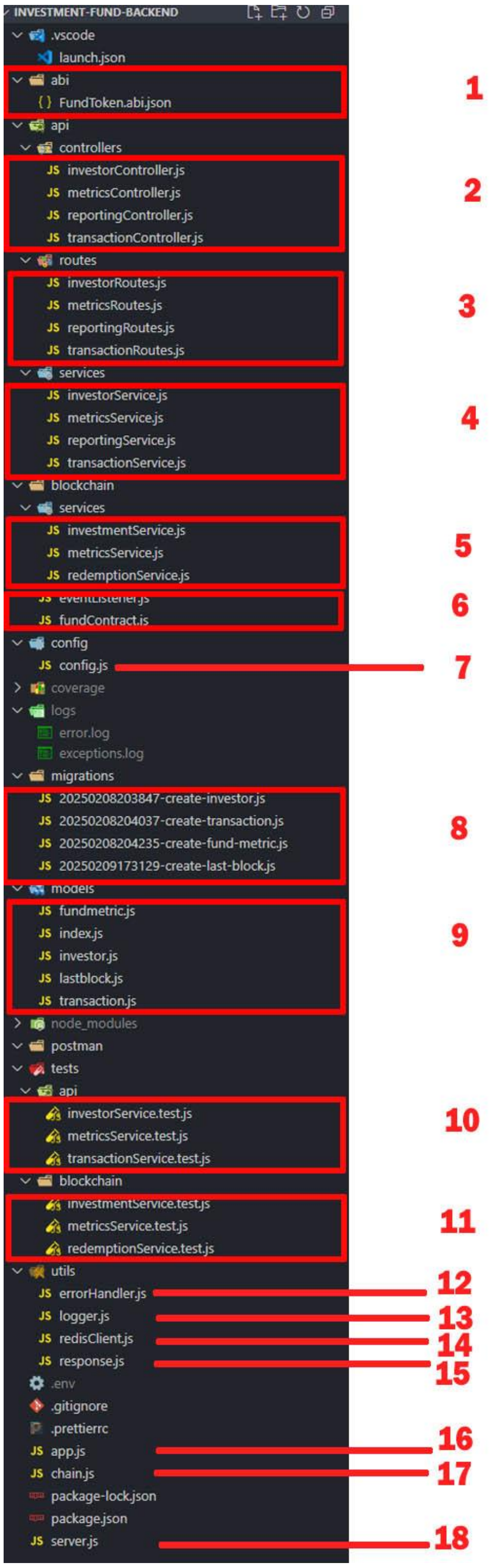
B. Components Development

There are 5 components as described in Architecture Diagram in this backend system.

A conventional project structure has been used in order to sufficiently implement the components architecture.

All the business logics are implemented as “services”, API logics as “routes” and “controllers”, Event Layer as a contract listener, Data Layer as “models” and Cache Layer as a “Redis” client.

Here is the project structure of my project.


```
INVESTMENT-FUND-BACKEND
 .vscode
   launch.json
 abi
   FundToken.abi.json
 api
   controllers
     investorController.js
     metricsController.js
     reportingController.js
     transactionController.js
   routes
     investorRoutes.js
     metricsRoutes.js
     reportingRoutes.js
     transactionRoutes.js
   services
     investorService.js
     metricsService.js
     reportingService.js
     transactionService.js
 blockchain
   services
     investmentService.js
     metricsService.js
     redemptionService.js
   eventListener.js
   fundContract.js
 config
   config.js
 coverage
 logs
   error.log
   exceptions.log
 migrations
   20250208203847-create-investor.js
   20250208204037-create-transaction.js
   20250208204235-create-fund-metric.js
   20250209173129-create-last-block.js
 models
   fundmetric.js
   index.js
   investor.js
   lastblock.js
   transaction.js
 node_modules
 postman
 tests
   api
     investorService.test.js
     metricsService.test.js
     transactionService.test.js
   blockchain
     investmentService.test.js
     metricsService.test.js
     redemptionService.test.js
 utils
   errorHandler.js
   logger.js
   redisClient.js
   response.js
 .env
 .gitignore
 .prettierrc
 app.js
 chain.js
 package-lock.json
 package.json
 server.js
```

1)	FundToken contract ABI
2)	API endpoint controllers
3)	API endpoint routes definition
4)	API services
5)	Smart contract event handling services
6)	Event Layer
7)	Sequelize configuration file
8)	DB migrations
9)	Model definitions
10)	Unit tests for API services
11)	Unit tests for event services
12)	Global error handler
13)	Logger (errors to file)
14)	Redis DB client
15)	Standardized response generator
16)	Express application
17)	Entry file for blockchain event layer
18)	Entry file for API server

Business Logics Layer: 4, 5
API Layer: 2, 3, 16
Data Layer: 9
Cache Layer: 14
Event Layer: 1, 6

C. Unit Tests

Unit test scripts are written against the Business Logics Layer, i.e. the service files.

D. Error Handling

All errors and exceptions are logged into logs/error.log and logs/exception.log file in a beautified style using the winston node module.