TED UNIVERSITY

CMPE 492 Low Level Design Report

CryptooFun

Team Members:

Kayra POLAT - 1000306178

Baturalp KIZILTAN - 4456996054

Emrecan ERBAY - 4221160055

Can ŞENGÜN - 1179712534

Supervisor:

Yücel ÇİMTAY

Jury Members:

Tolga Kurtuluş ÇAPIN

Emin KUĞU

Table of Contents

1 Introduction

Cryptocurrencies have become the most used investment tool in recent years. According to 2022 data, there are currently 300+ million users actively investing in cryptocurrencies. The global crypto market cap is $1.06 trillion as of August 1, 2022. It is in the hands of users to use this investment tool, which has a very bright future, correctly and safely. It is not right to invest in crypto money without understanding the risks and dynamics of the market. At this point, the CryptooFun application appears.

CryptooFun is ensure that people make minimal losses from their future investments by practicing with our application before investing their money in cryptocurrency exchanges. It will also provide real stock market experience by receiving the data of cryptocurrencies live. Thus, the experiences that people will have, will be more suitable for real life. The feature that distinguishes our project from other projects in the market is that we will enable people to compete while improving their knowledge with practice. Thus, we will encourage people to use our app.

In this report we will elaborate on the structural details of the project. We will talk about the packages that will be used to build the application and we will go into the details of our microservice structure and class interfaces.

1.1 Object Design Trade-offs

1.1.1 Speed vs Complexity

Since data needs to be received in real time, the speed of data processing and presentation is an important factor. In this case, technologies that provide faster data processing but are more complex can be chosen. However, simpler but slower technologies can offer an easier development process. Given these two aspects, it is more important to show live data in real time. Since users' ability to transact with real-time data is the cornerstone of the project, some of the complexity of the technology to provide and process the data can be sacrificed.

1.1.2 Ease of communication between microservices vs Performance

Using a microservice architecture in the project requires different functions to run on different services. This makes the project more scalable, but communication between services can become complex. It is inevitable that communication between services will affect performance. However, it is the developers' duty to minimize the performance loss. For this we will use the gRPC protocol, which is widely used for communication between microservices majorly due to its high performance.

For this part, it is worth going into detail about the microservice structure. Since each part of the application is embedded in a microservice, there are advantages and disadvantages to this approach. Advantages include the ability to scale each service separately, a problem in one service does not affect other services, and the division of labor within the development team. However, the disadvantages include the fact that each service can use different technologies, which can lead to integration problems, more management and monitoring, and debugging and troubleshooting difficulties.

1.1.3 Scalability vs Design Complexity

The aim of the project is to provide real-time cryptocurrency data, allowing users to make buy and sell transactions. However, if the project becomes popular, it may need to be scalable quickly. Therefore, the scalability of the project should be considered in the early stages of design. However, designing for scalability can increase the complexity of the project.

1.1.4 Flexibility vs Performance

If a problem occurs while using the Binance API, data will continue to be pulled from another cryptocurrency exchange. However, while this increases the flexibility of the application, it may reduce performance. If there is a problem with Binance servers, we will obtain data from another provider while minimizing the performance loss during the transition. However, it is very important that there is no data loss during the migration.

1.1.5 Security vs User Experience

The application allows users to make unreal buy and sell transactions. This can pose a significant security risk between users. Security can be ensured by correct authentication of users and data entry controls. However, these controls can negatively impact the user experience. We want to maximize both security and user experience by using the Identity Provider system for user authentication.

1.1.6 Buy-sell vs Real User Experience in Real Markets

The ability for users to trade with non-real balances allows the app to be used as an educational tool, giving them the opportunity to experience the risks associated with using real money. However, the disadvantages of this approach include the fact that real-time data cannot be manipulated, so the experience may be different from real markets, and users may gain less trading experience compared to trading with real money. The people who will use this app will certainly not be able to manipulate the markets. However, since our target audience is people with a small amount of money, people who do not know the cryptocurrency markets, and people who cannot manipulate even if they trade in real markets because they have a small amount of money, the issue of market manipulation is an issue that can be ignored.

1.2 Interface Documentation Guidelines

The interface, class and module structure will be as in the example below. First there will be the class name and description and then the name and description of the methods belonging to the class.

In case of naming conventions, the guideline for each programming language ecosystem is followed accordingly. For instance, Java suggests that "*class names should be nouns, in mixed case with the first letter of each internal word capitalized*"[1] by convention. We planned to dominantly make use of Java and JavaScript programming languages in our project and the guidelines for these two languages are listed respectively:

- Java naming conventions
- JavaScript naming conventions

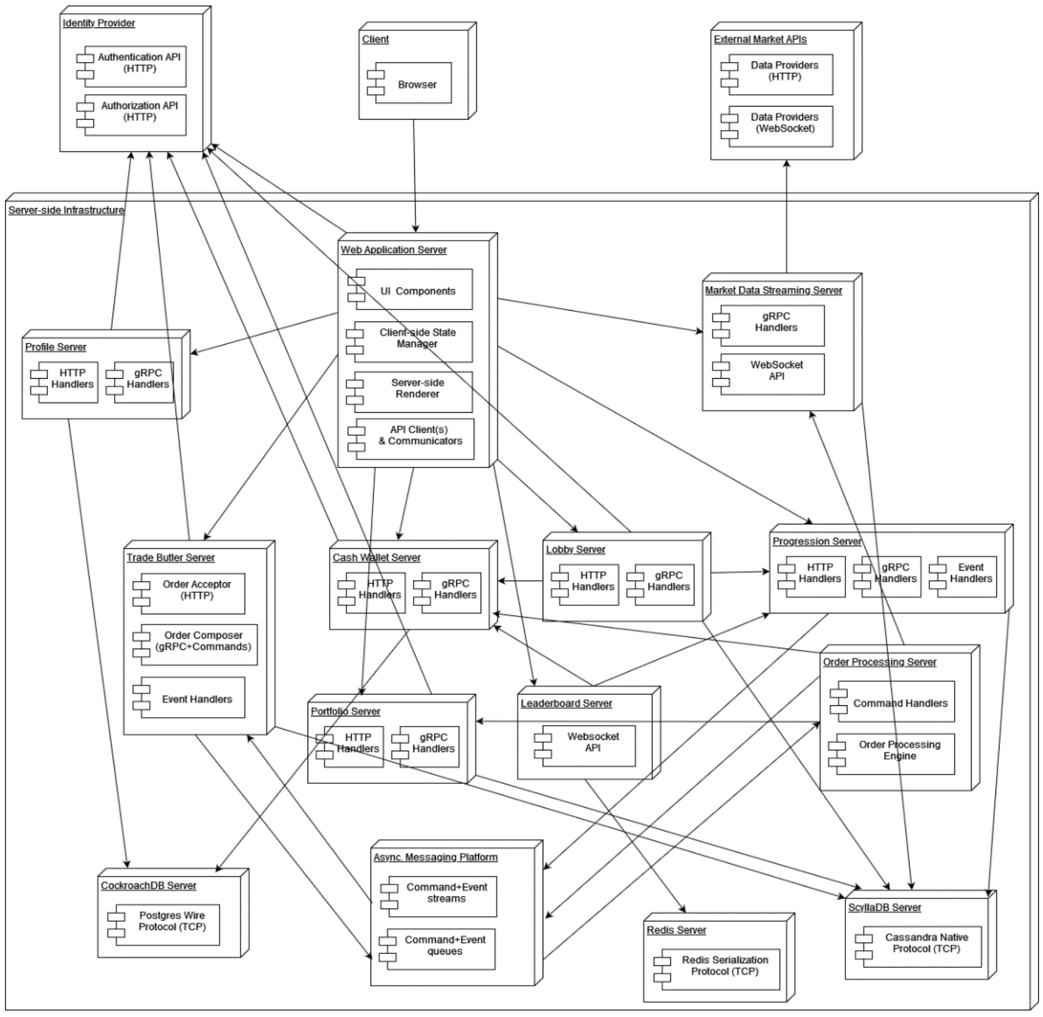
Example format:

className: *Description*

methodName: *Description*

methodName: *Description*

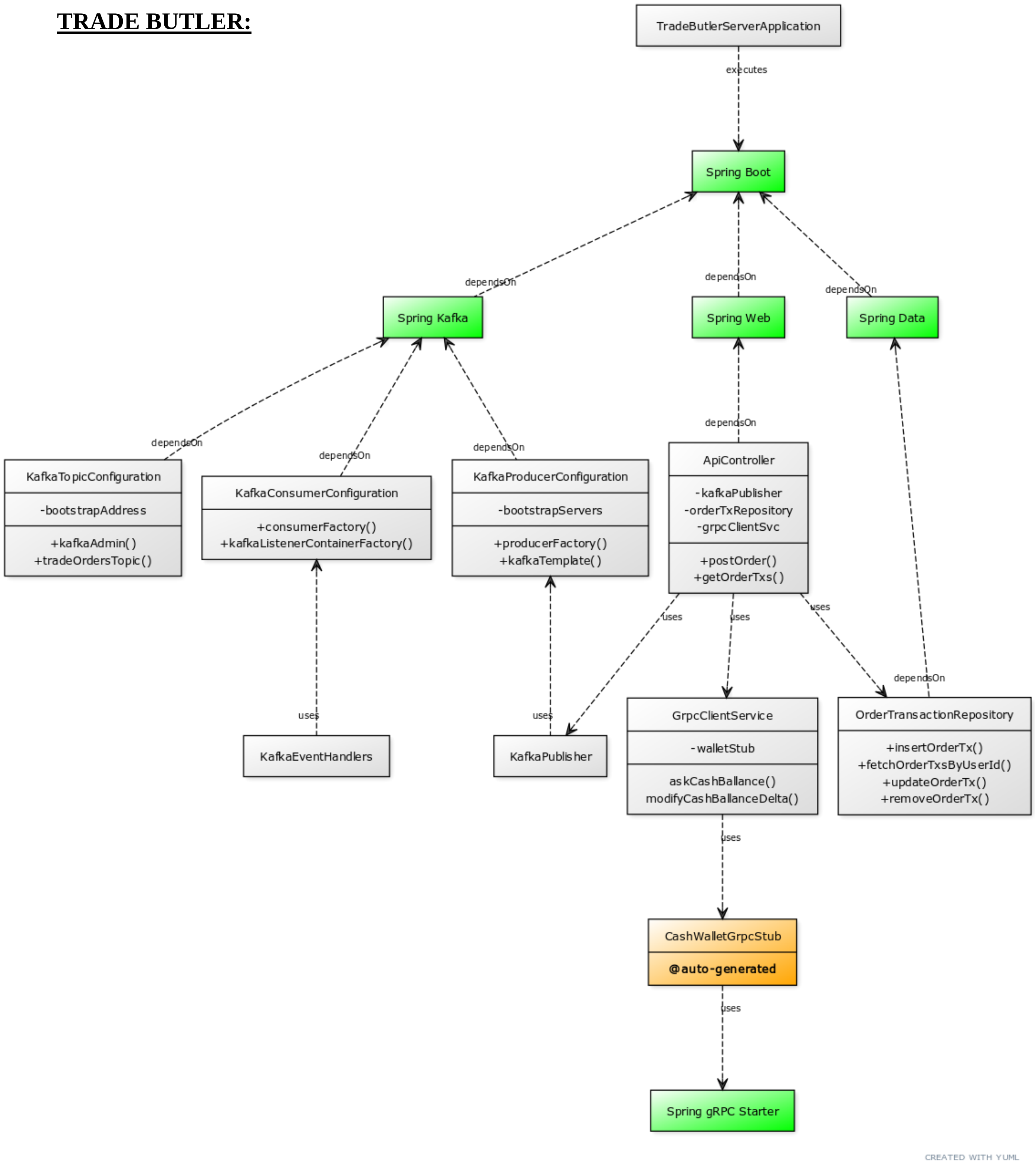
1.3 Engineering Standards (e.g., UML)



(Figure 1)

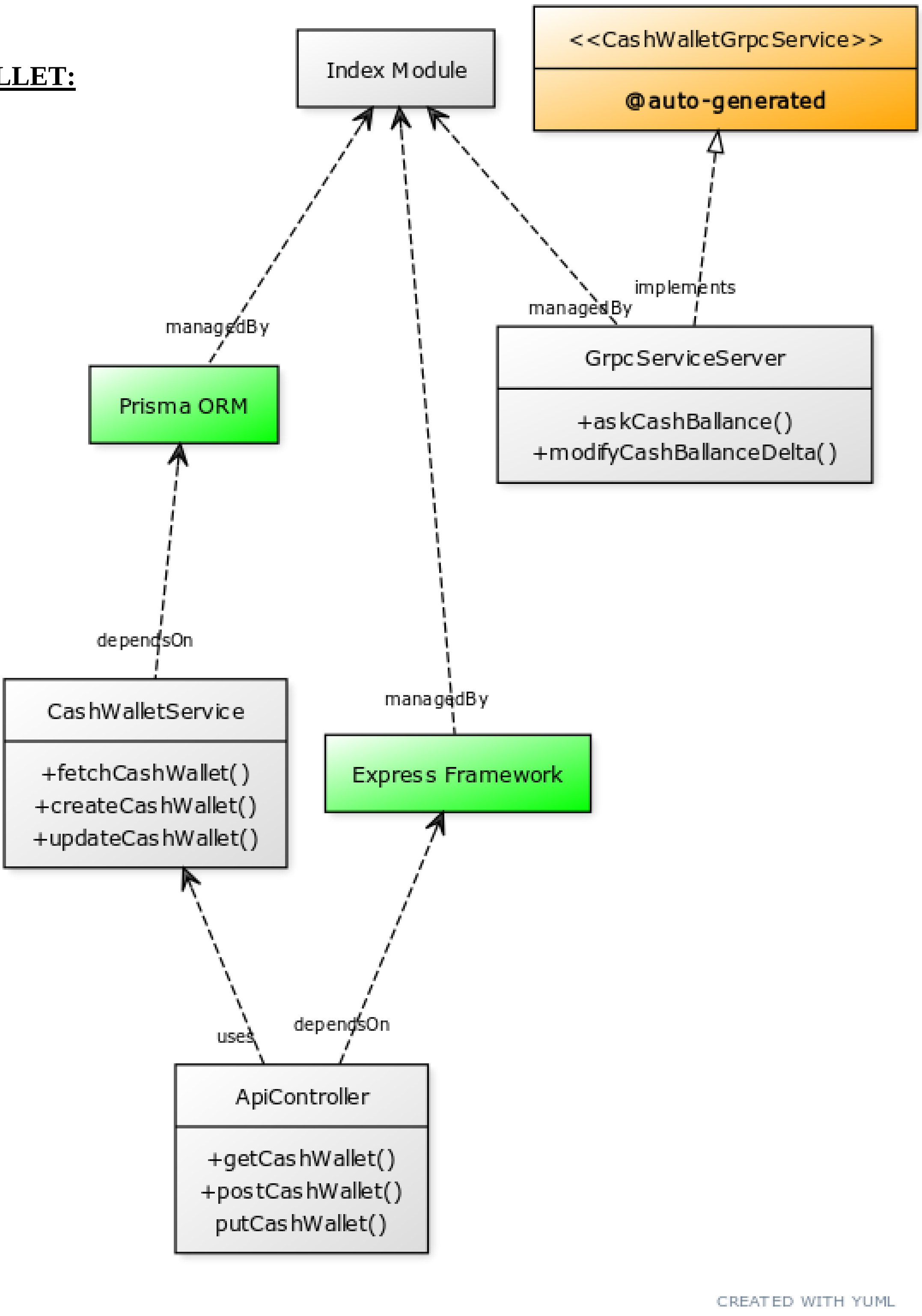
Since microservice architecture is used, each service will have classes and methods that have their own functions. As you can see in the picture, since there will be many microservice structures, we have created the classes and methods of only two services. One for building with Java spring boot and one for building with express.js. The picture below is the UML diagrams of these two services. We will go into the details of these classes in the Class Interfaces section.

<u>TRADE BUTLER:</u>



(Figure 2)

CASH WALLET:



(Figure 3)

1.4 Definitions, acronyms, and abbreviations

- **YAML:** A human-friendly, serializable language. Mainly used for defining configuration and specification files in the software industry.
- **Buf CLI:** A 3rd party command-line tool for linting and generating Protocol Buffers schemas.
- **POJO:** Plain Old Java Object
- **JSON:** JavaScript Object Notation
- **Monorepo:** A source code management strategy/approach, in the context of version-control systems such as Git. All kinds of services, scripts, apps, and modules are shared within the same repository.
- **Maven:** Apache Maven – an industry-standard build automation tool for Java projects.
- **NPM:** Stands for *Node Package Manager*. Became de facto standard for managing dependencies of Node.js projects.
- **Turborepo:** A build system for JavaScript and TypeScript projects. Provides capabilities and functionalities similar to Apache Maven for Node.js projects, especially in terms of monorepo-oriented dependency management strategies.

2 Packages

1. **Web Application** (~/apps/web/)

The web application is a Next.js project, written with React library in JavaScript. For styling purposes Tailwind CSS framework is utilized. The web app has its own self-contained executable/entry-point and can be deployable independently. Therefore, it’s put under “apps/web” directory of the project’s root. The app, which is part of the UI subsystem, serves as user-facing side of the project (front-end) and directly affects overall user experience.

2. **Protocol Buffer Definitions** (~/protobuf/)

The *protobuf* directory includes Protocol Buffers based data structures and gRPC service definitions alongside with a YAML configuration for Buf CLI. The protocol helps to define binary-serializable, cross-platform, language-neutral data structures and services. By that way, we can benefit from multiple programming language interfaces with less hassle, because code-generation tools for Protocol Buffers automatically converts protocol definitions to POJOs and JSONs in our case. Then, we just override the auto-generated interfaces with language-specific implementations to enable gRPC services.

3. **Services** (~/services/)

As it was described as one of the project’s goals, the back-end services are heavily based on microservices architecture. Since we follow monorepo-oriented approach, all services lay under the same source code repository alongside with the web application. We have already mentioned that we might benefit from the use of multiple programming language ecosystems in the previous parts and reports. Thus, we decided to go with Java and Node.js ecosystems. As a part of the monorepo structure, each language/runtime environment has its own root directory. By that way, we can share modules, 3rd party library packages, and commercial off-the-shelf products within the same language environment. For example, common Java packages and Spring Framework dependencies are shared across by defining a Maven root module under the “~/services/java/cryptoofun” directory. Similar concept is also applied to Node.js services via Turborepo based NPM workspaces.

A. **Java** (~/services/java/cryptoofun)

- **genproto:** *genproto* module contains auto-generated Protocol Buffers and gRPC code specific to Java.
- **Market Data Streaming:** An abstraction layer over external cryptocurrency APIs, e.g., Binance API. The service fetches historical data on demand and provides live data in regular intervals.
- **Trade Butler:** A receiver for new user orders. Sends orders to the dedicated Kafka topic to be processed later by other service(s).
- **Order Processing:** Consumes and processes trade orders.
- **Progression:** Manages achievements, awards, and experience levels for users.

B. **Node.js** (~/services/nodejs/cryptoofun)

- **genproto:** *genproto* module contains auto-generated Protocol Buffers and gRPC code specific to Node.js (JavaScript and TypeScript).
- **Profile:** A service, which is dedicated to manage user profile information. Collabarates with the Identity Provider infrastructure.
- **Cash Wallet:** Manages cash ballance for users.
- **Portfolio:** Manages cryptocurrency holdings of users.
- **Lobby:** Manages creation of lobbies and lobby sessions.
- **Leaderboard:** Maintains leaderboard table(s).

3 Class Interfaces

TRADE BUTLER SERVER APPLICATION:

ApiController: *Spring Web annotated REST controller class.*

public Response postOrder (OrderRequest order): *A REST controller for creating new trade orders.*

public Response getOrderTxs (String userId): *A REST controller for fetching order transactions by user ID.*

KafkaPublisher: *A KafkaTemplate dependency-injected component class for publishing messages.*

public void composeOrder (OrderMessage order): *A Kafka handler function for publishing trade orders.*

KafkaEventHandler: *A KafkaTemplate dependency-injected component class for consuming messages.*

public void handleOrderProcessed (OrderProccessedEvent event): *A Kafka handler function for consuming OrderProcessed events.*

KafkaProducerConfiguration: *A configuration annotation class for managing Kafka producer configurations.*

public ProducerFactory< String, TradeOrder> producerFactory(): *A Bean creator method for configuring a new Kafka publisher instance.*

public KafkaTemplate<String, String> kafkaTemplate(): *A method for creating KafkaTemplate Bean based on the new publisher.*

KafkaTopicConfiguration: *A configuration annotation class for managing Kafka topic configurations.*

public KafkaAdmin kafkaAdmin(): *An admin client instance for managing topics defined in the application context.*

public NewTopic tradeOrdersTopic(): *A method for creating a new Kafka topic with the name of "trade_orders".*

KafkaConsumerConfiguration: *A configuration annotation class for managing Kafka consumer configurations.*

public ConsumerFactory< String, TradeOrder> producerFactory(): *A Bean creator method for configuring a new Kafka consumer instance.*

public ConcurrentKafkaListenerContainerFactory<String, String>: *A method for creating KafkaTemplate Bean based on the new consumer.*

TradeButlerServerApplication: *A SpringBootApplication annotated class. Includes main entry-point for the service server application.*

public static void main (String[] args): *The main method for initializing the Spring Boot application.*

GrpcClientService: *A Service annotated class for managing and implementing gRPC client interface.*

public Cash askCashBallance(): *Asks cash balance to CashWallet service over gRPC client.*

public void modifyCashBallanceDelta(decimal delta): *Changes cash balance via CashWallet service over gRPC client.*

OrderTransactionRepository: *A Repository annotated class for managing and querying Order Transactions on demand.*

public OrderTx insertOrderTx(data): *Creates new order transaction.*

public OrderTx[] fetchOrderTxsByUserId(userId): *Fetches order transactions by user ID.*

public OrderTx updateOrderTx(txId, newData): *Updates order transaction by tx ID.*

public void removeOrderTx(txId): *Removes order transactions by tx ID.*

OrderTransaction: *A data model class for order transactions with basic getter/setters.*

CASH WALLET SERVICE:

Index: *Main module*

void main (): *Main entry-point.*

CashWalletService: *A module that contains business logic for manipulating CashWallet database model.*

CashWallet fetchCashWallet(id): *Fetches a specific cash wallet instance from database.*

CashWallet createCashWallet(data): *Inserts cash wallet instance into database.*

CashWallet updateCashWallet(id, newData): *Updates a specific cash wallet instance via database.*

ApiController: *REST-based HTTP controller module.*

getCashWallet(request): *Read CashWallet.*

postCashWallet(request): *Create CashWallet.*

putCashWallet(request): *Update CashWallet.*

GrpcServiceServer: *gRPC service implementation specific to the cash wallet service interface.*

Cash askCashBallance(): *Responses to cash balance requests over gRPC.*

void modifyCashBallanceDelta(decimal delta): *Processes changes to cash balance over gRPC.*

4 References

→ Fernandez, T. (2022, July 15). *What is monorepo? (and should you use it?)*. Semaphore. Retrieved December 25, 2022, from https://semaphoreci.com/blog/what-is-monorepo

→ Microsoft Docs Contributors. (2022). *Interservice Communication in microservices - azure architecture center*. Interservice communication in microservices - Azure Architecture Center | Microsoft Learn. Retrieved December 25, 2022, from https://learn.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication

→ Amazon Docs Contributors. (2022). *WebSphere Business Integration Pub/Sub Solutions*. Amazon. Retrieved December 25, 2022, from https://aws.amazon.com/pub-sub-messaging/

→ CMPE491 High Level Design Report – CryptooFun (2023)

→ *Spring*. Home. (n.d.). Retrieved March 22, 2023, from https://spring.io/

→ *Node.js web application framework - Express.js*. Express. (n.d.). Retrieved March 22, 2023, from https://expressjs.com/