**Crypto Trading Simulator Backend Documentation**

**Documentation Date:** May 14, 2025

**1. Overall Architecture and Purpose**

The provided codebase represents the backend system for a cryptocurrency trading simulator. The system is developed in Java and utilizes the Spring Boot framework for building RESTful APIs, managing dependencies, and handling configuration. The architecture follows a classic multi-layered approach, typically comprising:

* **Controllers:** Handle incoming HTTP requests, validate input, and delegate business logic to the service layer. These are located in sub-packages like com.example.cryptosim.account, com.example.cryptosim.holding, com.example.cryptosim.transaction, and com.example.cryptosim.crypto.
* **Services:** Contain the core business logic of the application. They coordinate interactions between repositories and other services. Examples include AccountService, HoldingService, TransactionService, and CryptoService.
* **Repositories:** Responsible for data access, abstracting the database interaction logic. They use Spring's JdbcTemplate for executing SQL queries. Examples include AccountRepository, HoldingRepository, and TransactionRepository.
* **Models/Entities:** Represent the data the system works with. These are divided into:
  + **Domain Models (utills.model):** Plain Old Java Objects (POJOs) used in the business logic and for communication with clients (potentially as Data Transfer Objects - DTOs).
  + **Database Entities (com.example.cryptosim.entity):** POJOs that directly map to the structure of database tables.
* **Converters:** Used for mapping between domain models and database entities.
* **Configuration:** Application settings, such as CORS and WebSocket configurations.
* **Utility Components:** Include functionalities like caching, ID generation, exception handling, etc.

The system appears to use a PostgreSQL database, judging by the SQL syntax for type definitions (e.g., CREATE TYPE transaction\_type AS ENUM) and data types (e.g., UUID, DECIMAL, VARCHAR, TIMESTAMP).

**2. Core Domain Models (utills.model)**

These classes represent the fundamental concepts within the application's domain.

* **Account**:
  + **Description:** Represents a user account within the system.
  + **Fields:**
    - id (String): Unique identifier for the account.
    - balance (Double): Current monetary balance of the account.
    - email (String): Email address associated with the account.
  + **Purpose:** Manages user account information and their balances.
* **Crypto**:
  + **Description:** Represents a cryptocurrency.
  + **Fields:**
    - symbol (String, final): The ticker symbol of the cryptocurrency (e.g., "BTC").
    - currentPrice (Double): The current market price of the cryptocurrency.
  + **Purpose:** Stores information about cryptocurrencies and their prices.
* **Holding**:
  + **Description:** Represents the quantity of a specific cryptocurrency owned by a user account.
  + **Fields:**
    - id (String, final): Unique identifier for the holding record.
    - crypto (String, final): Symbol of the held cryptocurrency.
    - quantity (Double): Amount of the cryptocurrency owned.
    - totalValue (Double): Total current market value of the owned quantity (quantity \* currentPrice).
    - accountId (String): Identifier of the account that owns this holding.
  + **Purpose:** Tracks the assets (cryptocurrencies) held by each account.
* **Transaction**:
  + **Description:** Represents a financial transaction (buy or sell) of a cryptocurrency.
  + **Fields:**
    - id (String): Unique identifier for the transaction.
    - type (TransactionType): The type of transaction (BUY or SELL).
    - crypto (String): Symbol of the cryptocurrency involved.
    - quantity (double): Amount of cryptocurrency transacted.
    - unitPrice (double): Price per unit of the cryptocurrency at the time of the transaction.
    - profitLoss (double): Realized profit or loss from the transaction (applicable on sell).
    - timestamp (LocalDateTime): The time when the transaction occurred.
    - accountId (String): Identifier of the account that performed the transaction.
  + **Purpose:** Records the history of all trading operations.
* **TransactionType (enum)**:
  + **Description:** An enumeration defining the possible types of transactions.
  + **Values:** BUY, SELL.
  + **Purpose:** Provides type safety and clarity when working with transaction types. *Note: The SQL schema defines transaction\_type AS ENUM('SELL','BUY'), which matches this enum. It was previously mentioned that TransactionType.java might also contain DEPOSIT, WITHDRAWAL, TRANSFER. These are absent in the currently provided TransactionType enum code, which only includes BUY and SELL.*

**3. Custom Exceptions (utills.model.exceptions)**

These classes define application-specific exceptions, all extending RuntimeException, making them unchecked exceptions.

* **FailedToLoadCrypto**:
  + **Description:** Thrown when there's an issue loading cryptocurrency data (likely from an external source).
  + **Constructor:** FailedToLoadCrypto(String message)
* **NotEnoughBalanceToBuy**:
  + **Description:** Thrown during a buy attempt if the account balance is insufficient.
  + **Constructor:** NotEnoughBalanceToBuy(String message)
* **NotEnoughHoldings**:
  + **Description:** Thrown during a sell attempt if the quantity of the held cryptocurrency is insufficient.
  + **Constructor:** NotEnoughHoldings(String message)
* **TransactionNotFoundException**:
  + **Description:** Thrown when a transaction is searched for by ID but is not found.
  + **Constructor:** TransactionNotFoundException(String message)

**4. Database Schema**

The database schema is defined via SQL CREATE TABLE statements, seemingly for PostgreSQL.

* **Table account**:
  + id (UUID, PRIMARY KEY): Unique identifier for the account.
  + balance (DECIMAL(20, 2), NOT NULL): Account balance.
  + email (VARCHAR(20), UNIQUE NOT NULL): User's email.
  + **Purpose:** Stores user account information.
* **Table holdings**:
  + id (UUID, PRIMARY KEY): Unique identifier for the holding.
  + crypto (VARCHAR(10), NOT NULL): Cryptocurrency symbol.
  + quantity (DECIMAL(20, 8), NOT NULL): Quantity held.
  + total\_value (DECIMAL(20, 2), NOT NULL): Total value of the holding.
  + account\_id (UUID, NOT NULL, REFERENCES account(id) ON DELETE CASCADE): Foreign key to the account table. Cascades deletes.
  + **Purpose:** Stores information about cryptocurrencies held by each account.
* **Type transaction\_type (ENUM)**:
  + CREATE TYPE transaction\_type AS ENUM('SELL','BUY');
  + **Purpose:** Defines an enumerated type in the database for transaction types.
* **Table transactions**:
  + id (UUID, PRIMARY KEY): Unique identifier for the transaction.
  + type (transaction\_type, NOT NULL): Type of transaction (BUY or SELL).
  + crypto\_symbol (VARCHAR(30), NOT NULL): Cryptocurrency symbol.
  + quantity (DECIMAL(20, 8), NOT NULL): Quantity transacted.
  + unit\_price (DECIMAL(20, 8), NOT NULL): Price per unit.
  + profit\_loss (DECIMAL(20, 2), NOT NULL): Profit or loss from the transaction.
  + created\_at (TIMESTAMP, DEFAULT CURRENT\_TIMESTAMP, NOT NULL): Timestamp of record creation.
  + account\_id (UUID, NOT NULL, REFERENCES account(id) ON DELETE SET NULL): Foreign key to the account table. Sets account\_id to NULL if the referenced account is deleted.
  + **Purpose:** Stores the history of all trading operations.

SQL TRUNCATE TABLE statements, wrapped in transactions, are also provided for clearing data from these tables.

**5. Application Components (com.example.cryptosim)**

**5.1. Account Management (com.example.cryptosim.account)**

* **AccountController (@RestController, @RequestMapping("/account"))**:
  + **Description:** Handles HTTP requests related to user accounts.
  + **Endpoints:**
    - GET /account: Retrieves account information (ID, balance, email).
    - GET /balance: Retrieves the current account balance.
    - PUT /balance/{quantity}: Updates the account balance (deposits or withdraws).
    - POST /reset: Resets the account (deletes transactions, holdings, and the account itself, then recreates it with initial values).
  + **Dependencies:** AccountService.
* **AccountService (interface Service, implementation AccountService @Component)**:
  + **Description:** Contains business logic for account management.
  + **Methods:** updateBalance, getBalance, resetAccount, getAccount.
  + **Dependencies:** AccountRepository, AccountConverter.
* **AccountRepository (@Repository)**:
  + **Description:** Manages data access for accounts in the database.
  + **Key Methods:**
    - createAccount(): Creates a new account (hardcoded email "[admin@gmail.com](mailto:admin@gmail.com)" and initial balance 100000.0).
    - getBalance(): Fetches the balance for "[admin@gmail.com](mailto:admin@gmail.com)". If the account doesn't exist, it's created.
    - updateBalance(Double balance): Updates the account's balance.
    - getAccount(): Fetches full account data for "[admin@gmail.com](mailto:admin@gmail.com)".
    - resetAccount(): Deletes transactions, holdings, and the account for "[admin@gmail.com](mailto:admin@gmail.com)", then recreates it.
  + **Dependencies:** JdbcTemplate, IUUIDConverter.
  + **Note:** The current implementation is strongly tied to a single, hardcoded user ("[admin@gmail.com](mailto:admin@gmail.com)").

**5.2. Holding Management (com.example.cryptosim.holding)**

* **HoldingController (@RestController, @RequestMapping("/Holding"))**:
  + **Description:** Handles HTTP requests for cryptocurrency holdings.
  + **Endpoints:**
    - GET /: Returns a list of all current holdings.
    - GET /{crypto\_id}: Returns holding information for a specific cryptocurrency by its symbol.
  + **Dependencies:** HoldingService.
* **HoldingService (interface Service - distinct from the one in account, implementation HoldingService@Component)**:
  + **Description:** Business logic for managing holdings.
  + **Methods:** createHolding, getAllHoldings, getHolding(UUID id), getHolding(String crypto), deleteAllHoldings, deleteHolding(UUID id), updateHolding.
  + **Dependencies:** HoldingRepository, IHoldingConverter.
* **HoldingRepository (@Repository)**:
  + **Description:** Data access for holdings in the database.
  + **Methods:** createHolding, getAllHoldings, getHolding(UUID id), getHolding(String crypto), updateHolding, deleteAllHoldings, deleteHolding(UUID id).
  + **Dependencies:** JdbcTemplate, IUUIDConverter.

**5.3. Transaction Processing (com.example.cryptosim.transaction)**

* **TransactionController (@RestController, @RequestMapping("/transactions"))**:
  + **Description:** Handles HTTP requests related to transactions.
  + **Endpoints:**
    - GET /: Retrieves all transactions.
    - GET /{id}: Retrieves a specific transaction by its ID.
    - POST /: Creates a new transaction. Input is validated using @Validated.
  + **Dependencies:** TransactionService.
* **TransactionService (interface Service - distinct, implementation TransactionService @Component)**:
  + **Description:** Core business logic for creating and managing transactions.
  + **Methods:**
    - createTransaction(Transaction transaction): Creates a transaction. If ID is not provided, it's generated. It then calls handleBuyTransaction or handleSellTransaction based on the type.
    - handleBuyTransaction(TransactionEntity transactionEntity): (private) Processes a buy transaction. Checks for sufficient balance, updates account balance, and updates/creates holding. Throws NotEnoughBalanceToBuy if balance is insufficient.
    - handleSellTransaction(TransactionEntity transactionEntity): (private) Processes a sell transaction. Checks for sufficient holdings, updates holding (or deletes it if quantity becomes zero), and updates account balance. Throws NotEnoughHoldings if holdings are insufficient.
    - getTransactionById(String id): Retrieves a transaction by ID. Throws TransactionNotFoundException if not found.
    - getAllTransactions(): Retrieves all transactions.
    - deleteAllTransactions(): Deletes all transactions.
  + **Dependencies:** TransactionRepository, HoldingRepository, AccountRepository, ITransactionConverter, Create (for HoldingFromTransaction).
  + **Logging:** Uses SLF4J for logging.
* **TransactionRepository (@Repository)**:
  + **Description:** Data access for transactions.
  + **Methods:** createTransaction, getAllTransactions, getTransaction(String id), deleteAllTransactions.
  + **Dependencies:** JdbcTemplate, IUUIDConverter.

**5.4. Cryptocurrency Information (com.example.cryptosim.crypto)**

* **CryptoController (@RestController, @RequestMapping("/api"))**:
  + **Description:** Provides information about available cryptocurrencies.
  + **Endpoints:**
    - GET /cryptos: Returns a hardcoded list of cryptocurrency symbols (e.g., "BTC", "ETH").
  + **Dependencies:** None explicitly shown, but relies on the hardcoded list.
* **CryptoService (@Component)**:
  + **Description:** Intended to fetch cryptocurrency data, likely prices, from external sources.
  + **Methods:**
    - formatUrl(String cryptoId): (private) Formats a URL for the Kraken API ticker endpoint (e.g., [https://api.kraken.com/0/public/Ticker?pair=%sUSD](https://api.kraken.com/0/public/Ticker?pair=%25sUSD)).
    - The actual method to fetch data and potentially use krakenWebSocketClient is not fully shown but implied.
  + **Dependencies:** KrakenWebSocketClient.
  + **Note:** The provided code for CryptoService is incomplete, showing only the URL formatting and constructor. It implies interaction with the Kraken API for price data.

**5.5. Data Converters (com.example.cryptosim.converters)**

This package contains interfaces and implementations for converting between domain model objects (used by services/controllers) and entity objects (used by repositories for database interaction). This is a common pattern to decouple application layers.

* **Interfaces:**
  + IAccountConverter: Defines methods convertToUser(AccountEntity entity) and convertToEntity(Account account).
  + IHoldingConverter: Defines methods convertToUser(HoldingEntity entity) and convertToEntity(Holding user).
  + ITransactionConverter: Defines methods convertToUser(TransactionEntity entity) and convertToEntity(Transaction user).
  + IUUIDConverter: Defines method convertFromString(String accountId) to convert a String to UUID.
* **Implementations (@Component):**
  + AccountConverter: Implements IAccountConverter. Uses IUUIDConverter.
  + HoldingConverter: Implements IHoldingConverter. Uses IUUIDConverter.
  + TransactionConverter: Implements ITransactionConverter. Uses IUUIDConverter. Handles potential IllegalArgumentException if UUID string is invalid.
  + UUIDConverter: Implements IUUIDConverter. Uses UUID.fromString().

**5.6. Entity Classes (com.example.cryptosim.entity)**

These classes represent the structure of data as stored in the database. Their fields typically mirror the columns of the database tables.

* **AccountEntity**: Maps to the account table. Contains id (UUID), balance (Double), email (String).
* **HoldingEntity**: Maps to the holdings table. Contains id (UUID), crypto (String), quantity (Double), totalValue (Double), accountId (UUID).
* **TransactionEntity**: Maps to the transactions table. Contains id (UUID), type (TransactionType), cryptoSymbol (String), quantity (double), unitPrice (double), profitLoss (Double), createdAt(LocalDateTime), accountId (UUID).

**5.7. Configuration (com.example.cryptosim.config)**

* **WebConfig (@Configuration)**:
  + **Description:** Configures web-related aspects, specifically CORS (Cross-Origin Resource Sharing).
  + **Bean:** corsConfigurer() (WebMvcConfigurer)
    - Allows requests from specified origins ([http://localhost:63342](http://localhost:63342/), [http://localhost:3000](http://localhost:3000/), [http://127.0.0.1:5500](http://127.0.0.1:5500/), "null").
    - Allows all HTTP methods (\*).
    - allowCredentials is set to false.
* **WebSocketConfig (@Configuration, @EnableWebSocket)**:
  + **Description:** Configures WebSocket communication.
  + **Method:** registerWebSocketHandlers(WebSocketHandlerRegistry registry)
    - Registers priceWebSocketHandler to handle WebSocket connections at the /prices endpoint.
    - Allows all origins (\*) for WebSocket connections.
  + **Dependencies:** PriceWebSocketHandler (implementation not provided, but it's the handler for price updates).

**5.8. Utility and Helper Components**

* **com.example.cryptosim.cache.PriceCache**:
  + **Description:** A simple in-memory cache for cryptocurrency prices.
  + **Implementation:** Uses a static ConcurrentHashMap to store prices, making it effectively a singleton.
  + **Methods:**
    - updatePrice(String symbol, double price): Updates or adds a price for a symbol.
    - getPrice(String symbol): Retrieves the price for a symbol, returning 0.0 if not found.
  + **Note:** This provides a basic caching mechanism. For a production system, a more robust caching solution (e.g., Caffeine, Redis) might be considered.
* **com.example.cryptosim.generators**:
  + IUUIDGenerator (interface): Defines a generate() method for UUIDs.
  + UUIDGenerator (@Component): Implements IUUIDGenerator, using UUID.randomUUID() to generate UUIDs.
* **com.example.cryptosim.create**:
  + Create (interface): Defines createHolding(TransactionEntity transactionEntity). This acts as a factory or creation helper.
  + HoldingFromTransaction (@Component): Implements Create. Creates a HoldingEntity based on data from a TransactionEntity (specifically a BUY transaction). It calculates the initial totalValue for the new holding.
  + **Dependencies:** IUUIDGenerator.
* **com.example.cryptosim.mapping**:
  + **Description:** Contains RowMapper implementations used by JdbcTemplate to map ResultSet rows to entity objects.
  + AccountRowMapper: Implements RowMapper<AccountEntity>. Uses IUUIDConverter.
  + HoldingRowMapper: Implements RowMapper<HoldingEntity>. Uses IUUIDConverter.
  + TransactionRowMapper: Implements RowMapper<TransactionEntity>. Uses IUUIDConverter. Converts the string "type" from the database to the TransactionType enum.

**6. Key Technologies & Frameworks**

* **Java:** Core programming language.
* **Spring Boot:** For rapid application development, dependency injection, auto-configuration, embedded web server (likely Tomcat), and REST API creation.
  + Spring MVC (for REST Controllers: @RestController, @GetMapping, @PostMapping, @PutMapping, @PathVariable, @RequestBody).
  + Spring Core (for DI: @Autowired, @Component, @Service, @Repository, @Configuration).
* **Spring JDBC (JdbcTemplate):** For simplified database interaction, managing connections, statements, and result sets.
* **Spring WebSockets (@EnableWebSocket, WebSocketConfigurer):** For real-time communication (e.g., price updates).
* **SLF4J:** Logging facade (used in TransactionService).
* **PostgreSQL (inferred):** Database system based on SQL syntax.
* **JUnit/Mockito (implied):** While not explicitly shown, these would typically be used for testing in a Spring Boot application.

**7. Design Considerations and Patterns (Inferred)**

* **Layered Architecture:** Clear separation of concerns into Controller, Service, and Repository layers.
* **Dependency Injection (DI):** Heavily used via Spring's constructor injection and annotations. This promotes loose coupling and testability.
* **Repository Pattern:** Data access logic is encapsulated within Repository classes, abstracting the data source.
* **Service Layer Pattern:** Business logic is encapsulated within Service classes.
* **Data Transfer Object (DTO) / Entity Separation:**
  + utills.model.\* classes act as domain models/DTOs used for API communication and service layer logic.
  + com.example.cryptosim.entity.\* classes are entities directly mapped to database tables.
  + Converter classes bridge these two representations.
* **RESTful API Design:** Endpoints are designed around resources (/account, /holdings, /transactions) using standard HTTP methods.
* **Custom Exception Handling:** Specific exceptions are defined for known error conditions, allowing for more granular error management.
* **Centralized Configuration:** Spring's @Configuration classes are used for CORS and WebSocket setup.
* **Factory/Creation Helper (HoldingFromTransaction):** The Create interface and its implementation encapsulate the logic for creating a HoldingEntity from a TransactionEntity.
* **Singleton-like Cache (PriceCache):** The static nature of PriceCache provides a simple, application-wide cache for prices.

**8. Example Data Flow (e.g., Creating a BUY Transaction)**

1. A client sends a POST request to /transactions with transaction details (type: BUY, crypto symbol, quantity, unit price, account ID) in the request body.
2. TransactionController receives the request. It validates the input (e.g., using @Validated).
3. The controller calls transactionService.createTransaction(transactionModel).
4. TransactionService converts the Transaction model to a TransactionEntity.
5. Since it's a BUY transaction, handleBuyTransaction is called: a. It fetches the AccountEntity using accountRepository.getAccount(). b. Calculates the total cost. c. Checks if account.getBalance() is sufficient. If not, throws NotEnoughBalanceToBuy. d. Updates the account's balance locally and persists this change via accountRepository.updateBalance(). e. Checks if a HoldingEntity for the crypto already exists using holdingRepository.getHolding(). f. If the holding exists, its quantity and total value are updated, and holdingRepository.updateHolding() is called. g. If the holding doesn't exist, create.createHolding(transactionEntity) (which is HoldingFromTransaction) is called to create a new HoldingEntity, and then holdingRepository.createHolding() persists it.
6. TransactionService then calls transactionRepository.createTransaction() to save the transaction details to the database.
7. The persisted TransactionEntity is converted back to a Transaction model.
8. TransactionController receives the created Transaction model and returns it in the HTTP response with a 201 Created status.