# Technical Documentation

**Author:** Alexandre Otávio Guedes Drummond

**Software Name:** Blockchain API

**Software Version:** 1.0

## Technical Information

This project has been developed not only considering software architecture best practices, including the utilization of SOLID principles for the overall architecture and REST for the API design, but also that this project is intended for testing purposes and is not currently expected to be deployed in a production environment. As a result, certain trade-offs were made to prioritize quick and efficient solutions that allowed for the application to be developed within the limited timeframe.

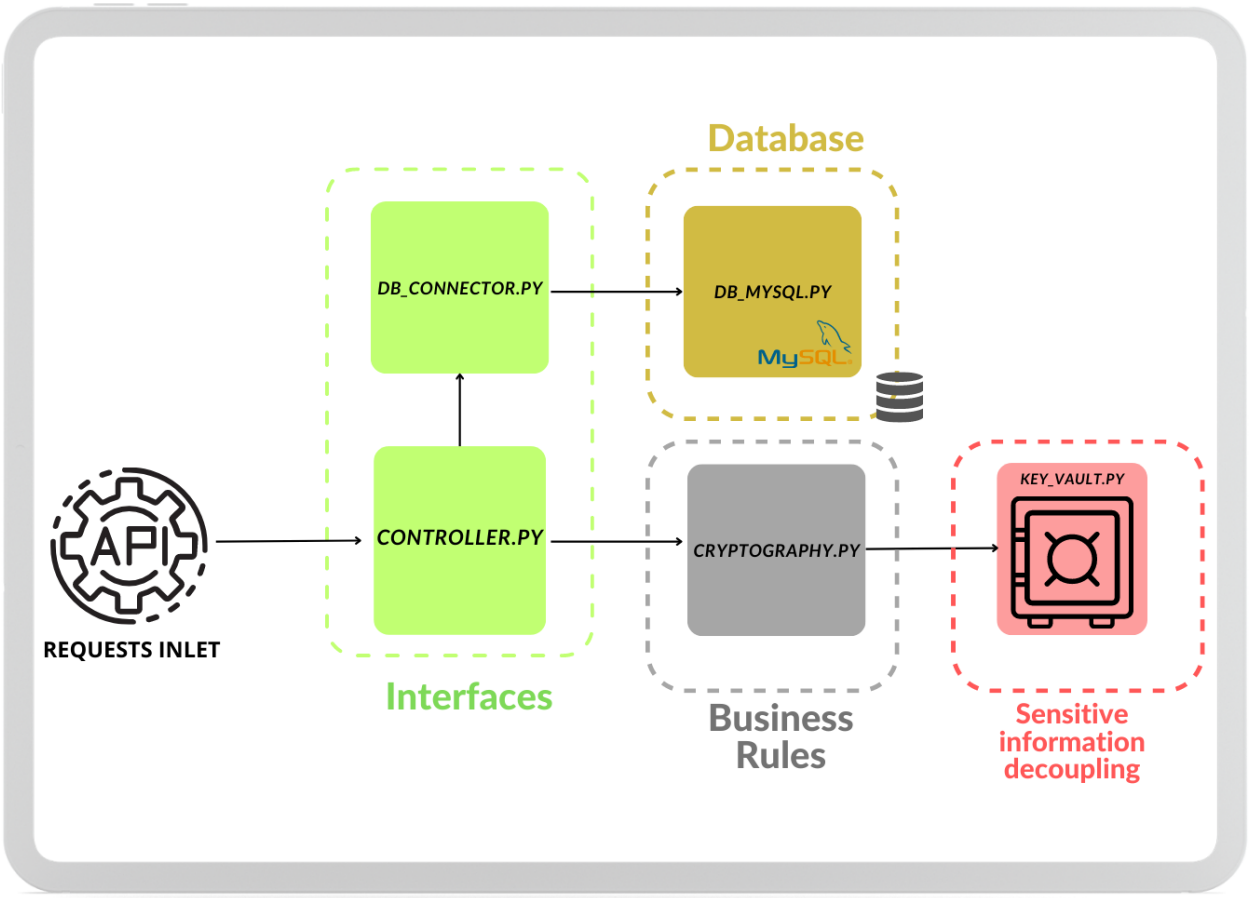
One area not prioritized for instance was security. Due to time constraints during the design phase, security was not prioritized, however, recognizing the importance of that aspect, a dedicated section has been included in this document to address this aspect.

All the components of the application have been designed with modularity and decoupling in mind, resulting in the diagram depicted in the next page. For example, although MySQL was chosen as the database solution based on the author's familiarity with it, it could easily be replaced with another SQL flavor by making modifications only to the "*db_mysql.py*" file. Similarly, the choice of the API framework can be altered by modifying the initial file, "*run.py*".

Numerous frameworks are available for developing APIs in Python, but Flask API was selected for its simplicity of implementation.

## Software structure

Despite being a small-scale system, the application has been designed in accordance with desirable software architecture principles. This approach ensures its scalability and facilitates maintenance. Consequently, the following structure has been reached:




- **run.py(REQUESTS_INLET):** API initializer and application entrypoint, containing the functions related to server's connection, as well as the endpoints.
- **cryptography.py:** Definition of the classes responsible for the rules of encryption, including the keys used during the generation, which are part of business rules.
- **key_vault.py:** Module to access the private key, stored externally to ensure its safety. As long this task is only a test it was built using AWS SDK to save to and AWS S3 bucket and load from it.

● **controller.py:** Responsible for interfacing the information exchanged between modules. It is required for achieving a satisfactory decoupling among components.
● **db_connector.py:** Abstraction of databases' task, this component interfaces the controller and MySQL functions. As long as this is a small project, this interface is redundant, it could be inserted in the controller, however on big projects those interfaces may become oversized so it is convenient the separation to emphasize what is related to the database.
● **db_mysql.py:** Contains MySQL functions that interact with the database.
● **.env:** Stores the environment variables, which improve the maintainability of the code through preventing that information to be hardcoded in the script. Security is not concerning since this is a test, therefore to save time it contains information that may be sensitive like, access keys, database host and password. However, it must NEVER be done in production.

**Database**

As the task required persisting a single information in the database, a single table named "*crypto_address*" has been created. However, its structure has been enhanced to incorporate key features that would be recommended for a production-ready application. The table structure is outlined below:

| **DATABASE CRYPTOCURRENCY_TASK** | | |
|---|---|---|
| **TABLE CRYPTO_ADDRESS** | | |
| **NAME** | **TYPE** | **DESCRIPTION** |
| **id** | *int* | Row unique identifier on table |
| **address** | *varchar(256)* | Address generated |
| **cryptocurrency** | *varchar(64)* | The symbol of the currency to which the address is related |
| **created_at** | *datetime* | Datetime of the creation |

In addition to the unique identifier for each address, requested to enable retrieval of specific rows through the API, an additional column was included to store the cryptocurrency symbol. This attribute proves valuable in a larger database, as it enables filtering of users' addresses based on the desired currency. Furthermore, a column was added to record the date and time of address creation, which proves highly useful for efficient database management.

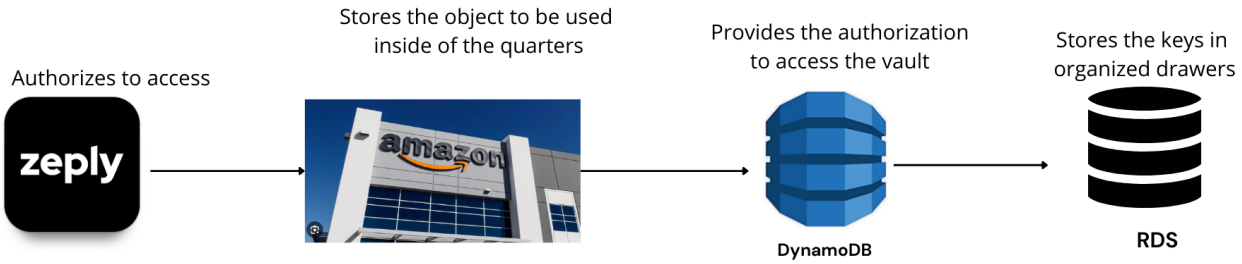
**Security**

As previously discussed, the optimal methodologies for ensuring the highest level of safety were not implemented due to the requirement of deploying additional components. In the current implementation, the private key is stored within an AWS S3 bucket, encapsulated as a JSON file. However, this approach falls short in terms of best practices, as it lacks supplementary layers of security.

In order to enhance the overall security measures, we propose the following solution: the ".env" file should exclusively store global variables to improve maintainability, eliminating any data that could potentially facilitate the recovery of sensitive information, such as credentials and URLs. Instead, variables like "db_host," "password," and "account_id" are stored within a non-relational database such as "DynamoDB," which is accessed through a dedicated API specifically developed for this purpose. Moreover, the credentials for this API are securely stored in a service like AWS Secret Manager or an equivalent, ensuring that only authorized applications within a defined context can access them.

Summarizing, to maintain a consistent structure private keys are stored in a designated relational database, with access credentials stored within the database accessed via the API. These credentials, in turn, are stored within the Secret Manager, which can only be accessed by authorized AWS applications.

An analogy  to real the security on the real world would be:




Zeply's customers are required to visit the company's office in person and present their identification documents as proof of their client status. Upon verification, they are issued a token by a Zeply attendant. This token serves as the authorization for accessing the AWS premises on behalf of Zeply. AWS headquarters, being from a large company, maintains robust security measures as surveillance cameras, alarms, and on-site guards, which surpasses the resources of smaller companies.

At the entrance of the building, the concierge carefully validates the token to grant authorization and provides an access card. The access card is used in the elevator to reach the designated floor where Zeply's customers are allowed (other floors are not allowed). On that

entrance, an employee verifies the customer's identity and provides them with a specific key to access an individual room within the vault. Inside this room, labeled drawers store all the customers' keys securely.

It is crucial to highlight that this project primarily focuses on the development of an API, which is most likely intended to serve as a tool for other software applications such as mobile apps or websites. Therefore, it is assumed that the initial security measures are already in place when utilizing this tool. User authentication and authorization are achieved through access to the app/website, which generates a token. This token needs to be included in the API's request body and is subsequently validated before any functionality is executed, ensuring a secure and controlled operation.

## Deployment

The entire project can be found on GitHub and accessed directly via the URL https://github.com/aogdrummond/blockchain-api. However, note that testing the project requires some environment preparation, such as creating a virtual environment, establishing a database connection, and setting environment variables. To simplify the deployment process, the project has been deployed using AWS API Gateway and is currently accessible at https://ke2pylckb0.execute-api.us-east-2.amazonaws.com.

For this deployment, the AWS service was chosen, integrating with Lambda Functions. The API Gateway routes incoming requests to these functions, which generate the responses returned to the requester. Consequently, some adjustments were made to each function (one function per endpoint). These adaptations are available in the GitHub repository branches labeled as "*deployed/list*" ,"*deployed/retrieve*" and "*deployed/generate*."

| ACCESS POINTS FOR THE API | | | |
|---|---|---|---|
| **Root URL** | https://ke2pylckb0.execute-api.use-east-2.amazonaws.com | | |
| **FUNCTION** | **METHOD** | **ENDPOINT** | **BODY** |
| Generate address | POST | /generate | {“crypto_currency”: “BTC”} |
| Retrieve from id | GET | /addresses/<address_id> | - - - |
| List all the addresses | GET | /list | - - - |

**(may be accessed using a tool like Postman or similar)**

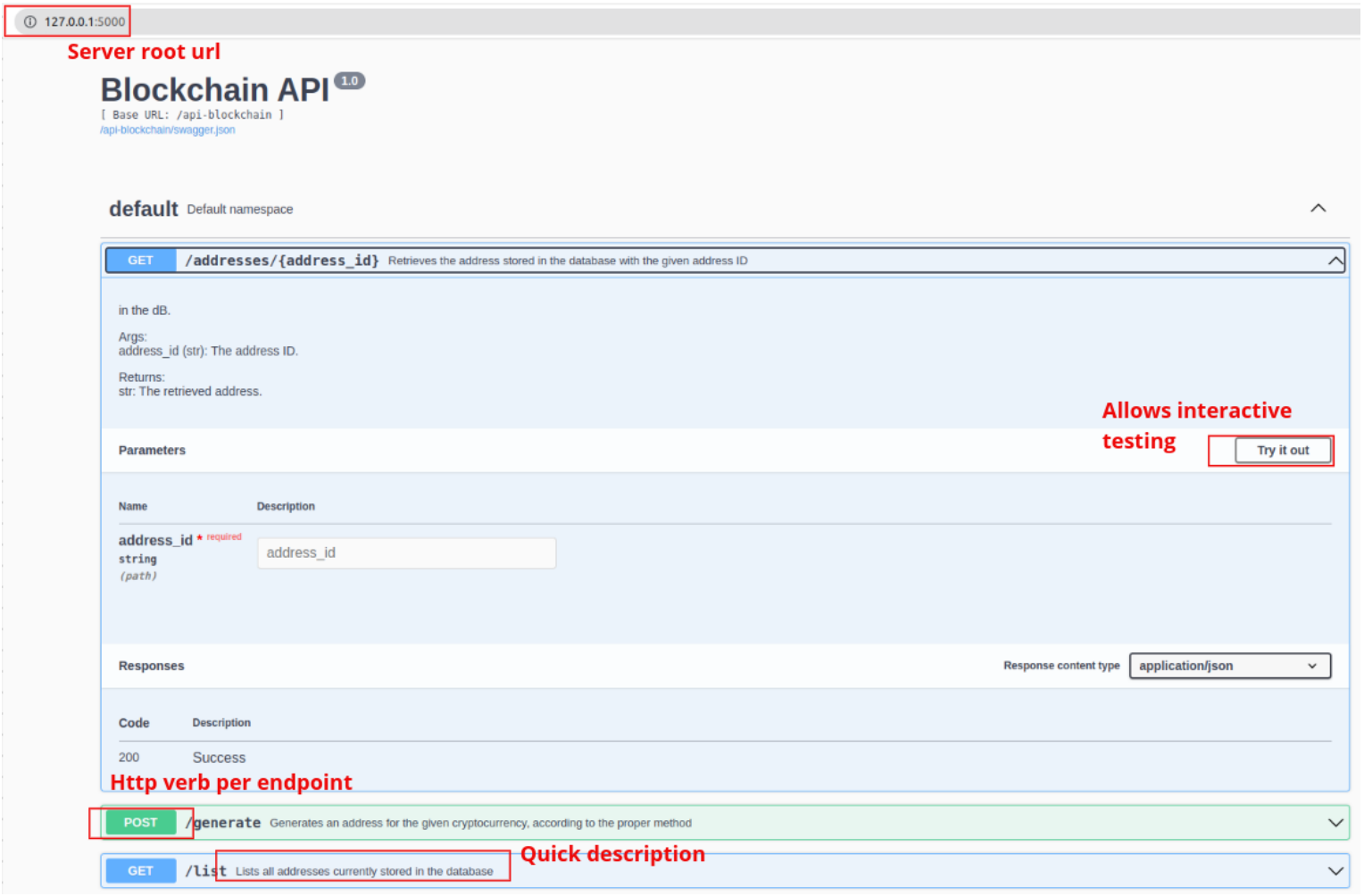
The decision to utilize API Gateway was primarily driven by its ease of implementation. However, it may not always be the optimal choice for production environments, as it requires additional attention from the DevOps team to handle the uploading, configuration, and management of numerous functions.

Alternatively, deploying the software as originally designed, utilizing Flask API, is possible but presents a higher initial complexity. This approach involves setting up a new server, but it offers also advantages. It allows for leveraging the full capacity of the Flask framework, such as the automatic deployment of Swagger, which facilitates API usage for developers. Moreover, it enables direct management of request loads and resource allocation for optimal performance.

Therefore, selecting the most suitable deployment approach requires careful consideration of various factors and trade-offs.

## Swagger

The primary reason for utilizing the "flask-restx" framework was to access the Swagger tool for automatic API documentation. This feature is seamlessly integrated with the API and can be accessed through the browser using the API's root URL. By utilizing Swagger, developers are provided with a convenient and user-friendly interface to explore and understand the implemented functionalities of the API. This eases the development process and facilitates the application of the API's features.




*Swagger available for interactive documentation*

## Testing

Automated tests are indispensable for ensuring the development of robust software. They play a crucial role in promptly detecting any newly introduced bug, providing confidence to programmers when making changes or additions. Consequently, incorporating automated testing is considered an essential aspect of professional software development.

The *pytest* framework was chosen to execute the automated tests due to the author's familiarity with the tool. The tests were categorized into three groups: unit, integration, and database testing.

Database testing: Since the software being developed is an API that inherently involves integration between various components, the database testing could be included as part of the integration tests module, but it was chosen not. It specifically focuses on testing the functions responsible for generating MySQL queries and analyzing their results. To ensure the isolation of the database during testing, a separate database with the same structure as the original one was created. The necessary details for connecting to this database are specified in the environment variables used during testing.

Unit testing: This category is designed to test the application of business rules, verifying the accuracy of each method declared in the "*cryptography.py*" module. The objective of unit testing is to isolate the objects being tested from any interference from other components. To achieve this, various functions are mocked, allowing for the examination of both successful and unsuccessful paths. Notably, the *pytest* functionality "*parametrize*" is employed to perform the same procedure with different inputs, such as different cryptocurrencies received. This enables vast testing of each scenario without bulking the module.

Integration testing: This component focuses on testing the API itself. It simulates the role of a client by verifying the responses received from executed requests. Successful execution of these tests requires the API to be running locally, as responses cannot be obtained without an active API instance.

To execute the tests after locally setting up the project, simply run the command from the root path of the project: *$ pytest tests*

### Dependencies

Since it was requested in the test statement no type of framework to be used, it was chosen to create a system with the smallest possible amount of external libraries. This option may be desirable in making the application more robust against conflicts due to dependency versioning. System's dependencies are presented in the following list:

| Package | Version | Package | Version |
|---|---|---|---|
| black | | mysql_connector_repackaged | 0.3.1 |
| boto3 | 1.26.135 | pysha3 | 1.0.2 |
| botocore | 1.29.135 | pytest | 1.0.0 |
| Flask | 2.3.2 | python-dotenv | 1.0.0 |
| flask-restx | 1.1.0 | Requests | 2.30.0 |
| coincurve | 18.0.0 | | |

## ChatGPT:

In order to optimize the time consumption during the task on various activities, the tool "ChatGPT" was utilized for specific tasks. This enabled the author to dedicate more time to finding solutions rather than being occupied with repetitive tasks such as formatting, writing docstrings, and defining type hints. Additionally, following the project development, the code was shared with ChatGPT to seek more concise and efficient solutions. The author carefully reviewed the outputs and implemented the suggested improvements while disregarding any suggestions that didn't provide an enhancement.