



University of
Zurich^{UZH}

Design and Implementation of an ICO Engine

*Vasileios Koukoutsas, Claude Müller, Simon Müller
Zürich, Switzerland*

Student ID: 16-718-991, 11-067-675, 12-715-389

Supervisors: Sina Rafati, Prof. Dr. Thomas Bocek,
Dr. Guilherme Sperb Machado

Date of Submission: April 12, 2019

Abstract

Recent developments in the digital world have led to a change in the way start-ups receive funding. Due the advent of Bitcoin and the subsequent rise of blockchain technology, so called Initial Coin Offerings (ICO) have become more and more relevant. In this master project, the open source ICO engine ICONator, based on an already existing engine from modum.io, was developed and polished. Existing components of the previous engine were transformed by focusing on reliability and robustness, while new components such as the one responsible for the Know Your Customer (KYC) process were introduced. The ICONator engine was then evaluated by observing software quality and the robustness of the system.

Die Innovationen und Neuerungen der letzten Zeit haben in der digitalen Welt zu einer Veränderung geführt, wie Start-ups zu Kapital kommen und neue Investoren finden. Der Erfolg von Bitcoin und damit einhergehend auch steigende Bekanntheit der Blockchain-Technologie haben das Sammeln von Kapital per Initial Coin Offering (ICO) mehr und mehr relevant gemacht. In diesem Master-Projekt wurde die Open Source Engine ICONator, basierend auf einer schon bestehenden Engine von modum.io, entwickelt und verfeinert. Bereits existierende Komponenten der bestehenden Engine wurden weiterentwickelt mit einem Fokus auf Zuverlässigkeit und Stabilität. Zusätzlich wurden komplett neue Komponenten wie zum Beispiel eine Unterstützung für den Know-Your-Customer (KYC) Prozess eingeführt. Zum Schluss wurde der ICONator in einer Evaluation auf Software-Qualität und Stabilität der einzelnen Komponenten untersucht.

Acknowledgments

We would like to thank our project supervisors and IConator project co-founders Dr. Guilherme Sperb Machado and Prof. Dr. Thomas Bocek and our supervisor Sina Rafati. Their guidance and mentorship throughout the project duration has been essential to its successful completion and our personal growth. Additionally, we would like to thank our families for their support during our studies.

Contents

Abstract	i
Acknowledgments	iii
1 Introduction	1
1.1 Motivation	1
1.2 Description of Work	2
1.3 Outline	3
2 Background and Related Work	5
2.1 Initial Coin Offerings	5
2.2 Existing ICO platforms	6
2.3 Requirements to the ICO engine	8
2.3.1 Functional	9
2.3.2 Non-functional	10
3 Architectural Overview	11
3.1 System Context	11
3.2 Platform Architecture	12
3.2.1 Message Queue	13
3.3 Source Code Structure	16

4 Applications	19
4.1 Core	19
4.1.1 Registration Steps	19
4.1.2 Core API	20
4.2 Email	22
4.2.1 Email implementation	22
4.2.2 Email templates	23
4.2.3 SMTP server	24
4.3 Monitor	24
4.3.1 Overview	25
4.3.2 Sale Tiers	26
4.3.3 Monitor instances and concurrency	29
4.3.4 Adding monitored addresses	30
4.3.5 Transactions	30
4.3.6 Transaction processing	32
4.3.7 Modeling of currencies and tokens	35
4.4 Rates	36
4.4.1 Fetching rates	36
4.4.2 Serving rates	37
4.4.3 Rates API	38
4.5 KYC	38
4.5.1 KYC Overview	38
4.5.2 KYC Controller	38
4.5.3 KYC Services	39
4.5.4 KYC Data	40
4.5.5 KYC API	40

5	Software Build and Deployment	43
5.1	Building	43
5.1.1	Local Development	43
5.1.2	Gradle	44
5.2	Containerization	44
5.2.1	Docker	44
5.2.2	Docker Compose	46
5.3	Cloud Deployment	48
5.3.1	Heroku	48
5.4	Operations	54
5.4.1	Key Generation	54
5.4.2	Key Consolidation	56
6	Software Evaluation and System Analysis	59
6.1	Software Quality	59
6.1.1	Code Coverage	60
6.1.2	Security	60
6.2	Robustness	61
6.2.1	Applications	61
6.2.2	System Context	63
7	Summary and Final Considerations	65
7.1	Summary	65
7.2	Final Considerations	66
7.3	Future Work	66
	Bibliography	69
	Abbreviations	73
	List of Figures	73

List of Tables	75
A Contents of the CD	79
B Sale tier configuration example	81
C Transaction processing flows	83
C.1 Pending transactions flow	83
C.2 Building transactions flow	85
D Heroku Bash Scripts	87
D.1 IConator Heroku Deploy	87
D.2 IConator Configuration Backup	88
D.3 IConator Configuration Update	89
E Heroku Configuration Variables	91
E.1 Core Service Configuration Variables	91
E.2 Email Service Configuration Variables	94
E.3 Rates Service Configuration Variables	98
E.4 Monitor Service Configuration Variables	101
E.5 KYC Service Configuration Variables	104

Chapter 1

Introduction

1.1 Motivation

The emergence of blockchain technology made it possible to create and distribute digital tokens. Tokens can have an inherent incentive for people to organize themselves around a certain purpose, even though they are scattered around the world and do not have to meet and trust each other. With the Bitcoin blockchain, that purpose is to have a platform that allows the participants to simply send and receive money. The token, in this case the Bitcoin, serves as a storage for value and is, at the same time, the means to pay for the usage of the platform. Here, paying for the usage means paying other participants for mining blocks. The token value is the incentive to take part in the network and ultimately contribute to it through the mining process.

With blockchains like Ethereum, tokens can be used for more complex applications than just money transfers. For example, it can be a token of admittance that allows the owner to use a distributed application. Such applications range from anything like distributed file storage to distributed computer games. These so-called utility tokens still have, next to their utilitarian value, a monetary value that is necessary as an incentive and is used for trading such tokens. Another kind of token is the security token. It is a digital form of financial securities.

Tokens can be used by companies in initial coin offerings (ICO) where they are sold and distributed to the public to raise starting capital. It is an alternative to initial public offerings, more conventional crowd funding or investors like hedge funds and venture capitalists. ICOs are currently very popular. \$6.3 billion have been invested via ICOs in the first 3 months of the year 2018 [1]. To run an ICO, one needs a platform that handles the on-boarding of investors and the distribution of tokens to investors once they have paid for them in fiat or cryptocurrency. The goal of this project is to design and implement such a platform and make it publicly available as open source software, free to use by anyone who wants to launch an ICO. The platform does not make any assumptions about the nature of a token sold in an ICO. The implementation of the token and the economic considerations in creating and selling a token are out of scope.

1.2 Description of Work

This report documents the design and implementation of an ICO engine. The term engine excludes any notion of front-end or user interface and sets the focus on the most crucial parts needed to perform an ICO. See Chapter 2 for requirements to the engine. The engine is build upon an already existing project going by the name of "ICOnator" [2]. It was initially conceived by Dr. Guilherme Sperb Machado who took a code base used for running the ICO of the company Modum [3], improved upon it and created an independent and open-source project from it. In this work, the main code structure of the ICOnator is retained and the existing source code is reused, extended, and improved. Important concepts and design choices of an ICO engine are elaborated and the results reflected back into the implementation.

This project contributes the following items to the ICOnator ICO engine:

- Automated health checking of critical engine parts.
- Addition of the concept of sale tiers.
- Total rework of blockchain monitoring and transaction processing including:
 - increased robustness against system crashes
 - increased levels of abstraction and extensibility
 - complex sale tier support
 - support for concurrently running monitor instances
 - well-defined and unified handling of monetary value
 - support for refunds
- Addition of a Know Your Customer (KYC) module that orchestrates the KYC process with an external KYC provider.
- Addition of the Elasticsearch, Kibana, Logstash (ELK) stack [4] for log analysis.
- Deployment of the ICOnator engine in a production environment using a public cloud provider (Heroku [5]):
 - deployed and configured ICOnator using 3 different methods:
 - * from source code [6],
 - * as Java Archive (JAR) files [7],
 - * as Docker containers [8]
 to discover the easiest and most cost-effective deployment method.
 - deployed and configured the necessary software component dependencies of ICOnator:
 - * RabbitMQ [9],
 - * Simple Mail Transfer Protocol (SMTP) server [10],

- * PostgreSQL database [11],
- * web front-end [12],
- * Bitcoin and Ethereum full nodes
- optimized Java Virtual Machine (JVM) performance to reduce operational cost
- Performed extensive end-to-end manual testing scenarios to discover bugs and corner cases using the Bitcoin and Ethereum testnets
- created Bash [13] automation scripts to deploy, configure and backup the ICONator engine in heroku.

Upon the start of this Master project, several design decisions were already made for the ICONator engine. Such design decisions are:

- Usage of Spring Boot [14] as the main application framework
- Following a modern software engineering approach by modularizing functionality into multiple modules which are loosely coupled and exchangeable.
- Communication between application modules via a message broker and message queues.
- Usage of Representational State Transfer (REST) for communication with external systems (e.g. a front-end).

All our contributed work is part of the ICONator project and therefore open-source, licensed under the Apache License version 2.0.

1.3 Outline

Chapter 2 Background and Related Work gives an introduction to the concept of an Initial Coin Offering, lists the existing ICO engines with a short description and enumerates the technical requirements that were set for the ICO engine.

Chapter 3 Architectural Overview gives an overview of the system components, both the ones that are part of ICONator and the ones that belong to the system context. A visual overview of the system architecture is provided with a detailed description of the messaging infrastructure, an integral component of the ICONator. Lastly, the source code structure is also described.

Chapter 4 Applications enumerates the 5 ICONator applications Core, Email, Monitor, Rates and KYC providing an overview and a more detailed documentation of their functionality, design, and API endpoints where applicable.

Chapter 5 Construction and Deployment provides all the necessary information for building, configuring, setting up, and deploying the ICONator and its needed system components either locally, using Docker-compose [15] or using a public cloud provider. An exhaustive list of the configuration variables for each app is provided with a short description and example values.

Chapter 6 Evaluation evaluates the results of the project, on one hand, through manually testing the engine and, on the other hand, by taking a step back and assessing the design and implementation decisions made. The system's code quality and robustness are analyzed.

Chapter 7 Summary and Final Considerations provides a short summary of the project and proposed improvements as future work.

Chapter 2

Background and Related Work

2.1 Initial Coin Offerings

The naming Initial Coin Offering follows the naming of a company's initial emitting of shares in an Initial Public Offering (IPO). In both cases, the goal of the offering is to raise capital, but apart from that they are inherently different. While in an IPO, shares of the company are sold, in an ICO, digital tokens are sold that can have more functions than simply representing a share of a company. Often they are also a means to participate in the product that should be created with the raised capital. The advantage of ICOs in comparison to IPOs is the small cost to run one and the low barrier for investors to participate. The downside is the legal uncertainty that one might face when running an ICO. Though, legal guidelines and regulation is catching up with the technological progress in that area.

The digital token sold in an ICO exists as a smart contract on a blockchain. They are allocated to investors on that blockchain via the smart contract. The smart contract maintains the balance sheet of the token distribution. An investor makes crypto or fiat currency payments and is allocated the corresponding amount in tokens. If the token's smart contract is already fully implemented, the investing, minting, and allocating of the token can be done directly on the blockchain. The investor simply has to make a payment to the corresponding smart contract and will thereby receive his tokens. This approach restricts the investment currency to the prevailing currency and maybe other tokens available on the blockchain in use. Another approach is to give investors the possibility of making investments on different blockchains and therefore offering multiple cryptocurrencies as a means of payment. Fiat currency can be included as well. The investor does not receive his tokens immediately but rather gets entitled to receive them in the future when the actual token minting is done on a specific blockchain. A legal framework for such a postponed allocation of tokens would be the Simple Agreement for Future Tokens (SAFT) [16]. Also of legal concern in an ICO is money laundering. To be compliant with money laundering law (depending on country) ICOs use the Know-Your-Customer (KYC) process to identify investors. An ICO operator should therewith make sure that his ICO is not used for money laundering activities. The KYC requirement does

challenge the scenario of running an ICO solely on basis of a smart contract. It would require an externally executed KYC process off-chain that plays back into the smart contract where it whitelists accepted investors.

2.2 Existing ICO platforms

This project is not the only attempt to create and provide an ICO engine or platform. A quick search on the Internet shows at least nine other providers of ICO platforms. So, why do we need another platform? What value can the ICONator add? To answer this we need to look at the platforms' properties.

The available products and providers vary in scope and provided services. They range from simply setting up an appropriate smart contract to comprehensive software and service packages. The latter products are all a mixture of software and manual services. In those cases the software is closed-source and running an ICO with these providers implies paying for the software and any service used.

The providers with the most comprehensive offers include the following things in their services:

- A customizable software platform that includes administrative control over the ICO, a user interface for investors.
- Consulting in the conception of the token and the token economics
- Technical support in the token creation
- Reviewing of the white paper
- Conducting or supporting the ICO's marketing
- Legal advice and support
- Conduct of KYC and anti-money-laundering (AML) processes

The providers that offer some or all of these services are ICO Malta [17], Eidoo [18], Das33 [19], Coin Factory [20], ICOBox [21], ICO platform [22], The ICO Platform [23] and Tokengate [24].

Providers like Eidoo and Das33 accumulate all their ICOs on the same platform. The benefit herein is that an investor has to go through KYC only once and can from then on invest in all ICOs that are running on the platform. The platform acts as a mediator between companies that conduct an ICO and the investors. With such unified platforms, the providers have set up a careful selection process in which companies have to apply with a white paper and other information about the company. This obviously leads to more trust by the investors that the available ICOs are not frauds.

The platform Das33 further provides a designated token (DasCoin) with their platform that is used as a security token by every company that wants to run an ICO with them. Companies can therefore not distribute their own tokens with Das33. This implies that participating companies have no need for a utility token and blockchain technology is probably only used for the funding but not for the actual product.

TokenGate is an example of a platform that is deployed separately for each ICO issuer. They provide back-end functionality and user interfaces for investor registration, KYC/AML, investor dashboard and administrator dashboard. An important part of TokenGate's offer is that by using the platform one can be sure that the ICO will be compliant to Swiss law.

The only product that we found that is free and open-source is called Token Wizard [25]. It is based on open-source Ethereum smart contracts provided by TokenMarket [26]. The web-based tool leads the user through the setup of an ICO that is solely based on a configurable smart contract. This includes things like sale tier setup and white listing of payment addresses and results in a configured smart contract. The contract can then be deployed on the Ethereum blockchain and takes over the ICO from there. It is clear that such a solution is limited to the basic technical functionality that an ICO requires. That is, investors can make payments directly to the contract and are allocated their tokens directly on the blockchain. The only available investment currency is Ether. The KYC process has to be handled separately and no user interface is provided for investors or administrators when the ICO is running.

Table 2.1 summarizes the properties of the mentioned ICO engines/platforms. The columns meanings are:

Name The name of the platform, engine or provider.

Open-source If the platform is open-source or closed-source.

Free If the platform is available for free or if it is a paid service.

Support If the platform provider offers, besides the basic ICO functionalities, additional support in technical, legal, or marketing matters.

Self-deployable If the platform or software can be set up and deployed by oneself on a chosen infrastructure.

KYC If the platform integrates or offers a KYC and AML solution.

Mediator If the platform acts as a mediator between investors and all the ICOs registered on the platform. Or, on the other hand, if the platform is a software package that is used to manage ICOs independently without any connection between them.

Fiat If the platform allows for investments in fiat currencies.

Our findings show that there is much room for an ICO platform that is free, open-source, controllable and configurable by the conductor of the ICO. Controllable in the sense that

Name	Open-source	Free	Support	Self-deployable	KYC	Mediator	Fiat
Eidoo	no	no	yes	no	yes	yes	yes
Das33	no	no	yes	no	yes	yes	no
ICO Malta	no	no	yes	no	yes	no	yes
Coin Factory	no	no	yes	yes	yes	no	yes
ICO platform	no	no	yes	no	yes	yes	no
The ICO platform	no	no	yes	no	yes	yes	yes
Tokengate	no	no	yes	no	yes	no	yes
ICOBBox	no	no	yes	no	yes	no	yes
Token Wizard	yes	yes	no	yes	no	no	no

Table 2.1: Overview of ICO platforms.

it can be deployed on a chosen infrastructure and run at command of the operator. Token Wizard is the only open-source solution found so far, but it is only a smart contract that needs to be deployed on the Ethereum blockchain. The vision for the IConator is to allow more flexibility. It is not just a smart contract but a blockchain-independent ICO engine. It integrates the KYC process and communications with the investors. In comparison to the commercial platforms, all the technical or legal consulting and support are missing in the IConator project because such services are not part of an open-source software product. The IConator also does not provide a ready-to-use investor or administrator interfaces. It leaves that up to the operator of the engine. Of course a customizable user interface would be a valuable future extension to the project. The value of the IConator, compared to the existing platforms/engines, lies in its open-source nature, customizability, extensibility, and in that it is free of charge.

2.3 Requirements to the ICO engine

This section lists the requirements that are posed to the IConator. They were not elicited with the help of a community or external stakeholders but were produced by the authors themselves, people that are involved in this project, and by the team behind the EUREKA token [27] and ScienceMatters project [28]. The EUREKA token ICO was planned to be the first opportunity to employ the IConator in a real-world use case. That is why it significantly influenced the requirements.

Since the product in mind is described as an engine and not as a whole platform, the following constraints become clear immediately. The engine does not include:

- a front-end for investors
- a front-end for administrators

- the actual token creation and underlying smart contract on a blockchain

But it should provide interfaces with which independently developed front-ends can interact. And in regard to the token creation the engine must store and provide all the information needed to later correctly distribute the tokens.

It was decided to not include support for fiat currencies because that would require partnering with another service provider for obtaining an interface to fiat currency transaction streams. Only the Ethereum and Bitcoin blockchain need to be supported. Though the source code should provide extension points, at which the engine can be extended with other blockchains.

The requirements are separated into functional and non-functional and are listed below.

2.3.1 Functional

- The engine must allow for registering for participation in the ICO.
- The engine must automatically inform investors about their registration.
- For each investor the engine must collect and store blockchain addresses including
 - an address that will be used as the source from which the investor makes his payments.
 - an address to which refunds can be paid.
 - an address to which tokens can be allocated to.
- The engine must distribute addresses (one per blockchain) to an investor to which he can make payments to.
- The engine must provide an investor with information about their payment and the token allocation or the entitlement for a refund.
- The engine must allow the operator to configure a sale tier structure that will be used in the ICO.
- The engine must be able to store, update and provide sale tier status information.
- The engine must continuously monitor supported blockchains and detect payments made to the distributed payment addresses.
- The engine must allocate tokens amounts according to the detected payment amounts, the currently active sale tier and the current exchange rate. This does not include the distribution of the tokens on a blockchain. The token allocation information must be available to the operator.
- The engine should regularly record and store exchange rates (from a defined base currency to supported cryptocurrencies) to ensure that exchanges rates are available for processing incoming payments.

- The engine must create refund entries for detected payments that led to an error while processing them. The refund information must be available to the operator.
- The engine must store additional information for processed payments, like exchange rate, block time or block number. This is needed for later verification if necessary.
- The engine must provide an interface for connecting it to an external KYC provider. The KYC process should be handled automatically from the viewpoint of the operator. This implies that the engine must automatically redirect the investors to the external KYC provider and obtain the results of the KYC process.

2.3.2 Non-functional

- The behavior of the engine must comply with privacy regulations. This comprises the processed and stored investor data and their payments. The engine should store a minimal set of information needed to provide all needed functionality. Though, in the end, handling the data according to privacy law is up to the operator.
- The engine must comply with money laundering regulations. The operator must ensure that an appropriate KYC process is integrated and used correctly.
- The engine must be robust against crashes of internal and external systems so that first and foremost no invested cryptocurrency amounts and allocated token amounts are compromised. The engine or its parts must be simple to restart after a crash and must commence its work without missing out on any payments.
- The engine must secure its communication from unauthorized eavesdropping and manipulation be it between engine sub-parts or between the engine and the system context. All data in transit and storage must be secured from unauthorized access and modification. Though, the operator of the engine has unrestricted access and modification rights to all data stored. The engine does not have to secure the data from malicious internal administrators.
- The engine should be extendable with new functionalities - like new interfaces or interoperability with additional blockchains - without the need for big changes in the already existing code parts.
- The engine should be composed of separable and exchangeable parts that are loosely-coupled.
- The engine should be easy to deploy. And the deployment should be flexible enough to allow for it to work on different platforms.
- The used methods for developing and constructing the software should pertain to modern software development standards and make use of well-known and widespread frameworks and libraries where needed.

Chapter 3

Architectural Overview

This chapter provides an architectural overview of the whole ICO platform and explains important considerations and decisions made. Section 3.1 describes the interactions between the IConator and its surrounding environment, while Section 3.2 describes the relationships between the decoupled components within IConator. Furthermore, in Section 3.3 the structuring of the source code is shortly explained.

3.1 System Context

The environment of the IConator consists of the front-end application that is used by investors and the external dependencies that are required for the IConator to operate correctly. The external dependencies are a single database that is used to store all the data, the message queue broker that handles the communication between the modules, the SMTP server that sends out all the emails, the currency exchanges used in the currency conversion of investments and the blockchain full nodes that are needed to detect payments. The system context diagram shown in Figure 3.1 illustrates this environment. In the following, each element of the system context is described in more detail.

Front-end Application IConator does not provide a front-end for the ICO. It needs to be created separately and can be connected to the IConator via the RESTful API provided by the modules. The API is described in detail in the corresponding module sections in Chapter 4.

Database IConator stores all its data in one single external relational database. All databases supported by Hibernate can be connected to IConator.

Message Queue Broker IConator is composed of several decoupled modules that need to communicate with each other. This communication happens over a message queue. IConator uses the Advanced Message Queuing Protocol (AMQP). The Message Queue Broker is described in detail in Section 3.2.1. How the interaction between the modules works is further described in Section 3.2.

SMTP Server IConator will send emails to investors after certain steps in the investment process are completed. Therefore, an external SMTP server needs to be set up to enable the email application within IConator to send emails.

Currency Exchanges Stable and trusted currency exchanges are a key part in ensuring a successful ICO. The rates module within IConator uses the exchange rates fetched from the currency exchanges to compute the value of the invested cryptocurrency in fiat currency. In order to ensure a more robust operation, multiple currency exchanges should be used. There is also a fallback mechanism in place if rates could not be fetched for some time interval. See Section 4.4 for detailed information.

Blockchain Full Nodes In order to compensate investors for their investment, IConator needs to be able to detect all the payments that were made in cryptocurrencies. This requires the connection to respective full nodes of the supported blockchains that need to be up-to-date all the time. The monitor application within IConator is constantly monitoring the blockchain full nodes and checking for newly received payments.

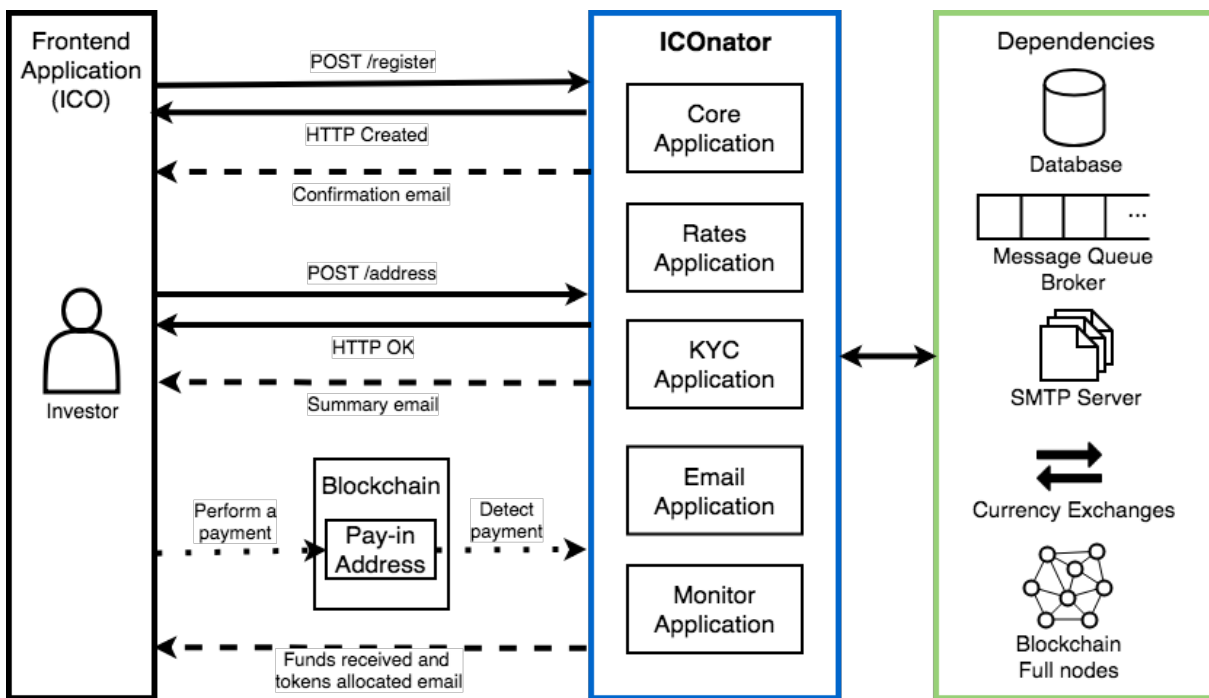


Figure 3.1: System Context of IConator. Source: figure adapted from Dr. Guilherme Sperb Machado's presentation.

3.2 Platform Architecture

The IConator platform is essentially a collection of multiple modules that communicate with each other over a message queue. How the message queue works is described in Section 3.2.1 in detail. As can be seen in the platform architecture diagram shown in

Figure 3.2, each module is an independent building block and together they form the IConator platform. Decoupling the application into these smaller pieces of software makes it easier to develop, maintain and deploy the platform. Every module can be deployed separately from each other, with the core module acting as the 'entrance' to the platform. Each module has its own set of clearly defined responsibilities which are also reflected in the module names.

The core application acts as the entry point of the IConator and exposes most of the API endpoints that are available. It has various functionalities described further in Chapter 4 but mainly handles the registration of new investors and persists them in the database.

The email application is the only application that does not need any access to the database. It is also the only application communicating with the external SMTP server, as it needs the server to send out emails.

The monitor application is the only service with a connection to the blockchain full nodes, while the rates application is the only one communicating with external exchange services.

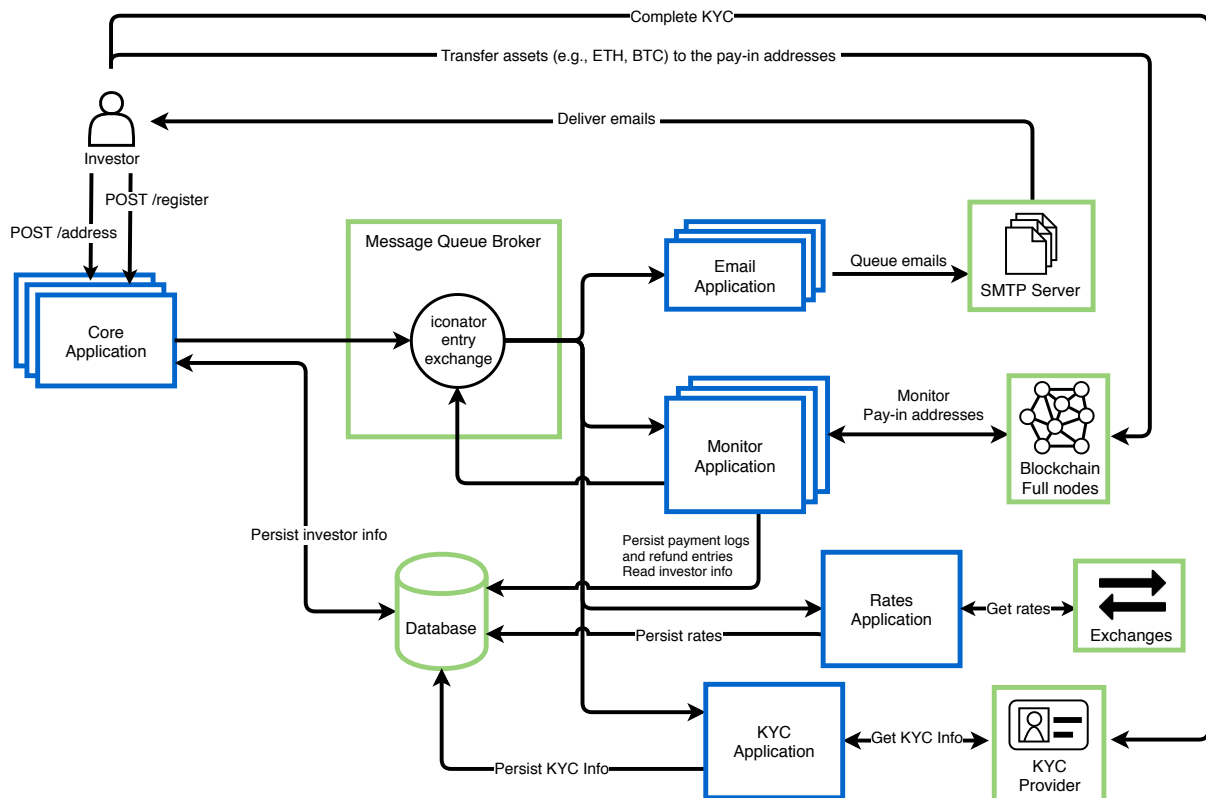


Figure 3.2: Platform architecture with simplified message queue. Source: figure adapted from Dr. Guilherme Sperb Machado's presentation.

3.2.1 Message Queue

A message queue is a queue that is used for communication between decoupled services needing to communicate with each other. A service creates a message containing the

information it wants the other service to know. The message is then put in the message queue. The message is stored on the queue until a message consumer processes the message. Each message is only processed once. If the message consumer encounters an error while processing a message, the message is queued again. The application adding a message to the queue is called the publisher, while the application processing the message is called the consumer [29].

Advantages of message queues

There are several advantages a message queue can bring to a decoupled system over traditional direct communication between the services:

Performance When communicating over a message queue, the services do not interact with each other directly. Each service only interacts with the queue. This allows for a module to add a message to the queue without needing to wait for the receiving end to process the message. This asynchronous message flow provides optimal performance as no module ever has to wait for the consumer to fetch the message [30].

Scalability Message queues make it easy to scale a system. If, for example, multiple instances of the monitor application need to be deployed they can all write to the same message queue without risking a collision between messages. If there are a lot of messages waiting in the queue to be processed, it is also possible to increase the number of consumers. This will speed up the operation as multiple messages can be consumed in parallel [30].

Reliability As each message is stored in the queue, the message queue is able to persist all messages that have not yet been processed. This leads to a reduction of communication errors in case one or several modules of IConator go offline. The use of a message queue dramatically increases the reliability of the platform without the need of adding specific code handling such cases [30].

Message queues in IConator

IConator uses the Advanced Message Queuing Protocol (AMQP) as its protocol for the message queue. AMQP is an open standard [31]. IConator uses the RabbitMQ implementation of the AMQP standard.

Every type of message that is sent between one of the decoupled modules uses its own message queue. I.e. there are no queues which listen to multiple routing keys. Therefore, there exist various different message queues of which each fulfills a different purpose. All the routing keys used by the queues in the IConator platform are described in Table 3.1 including the purpose of the queue and its publishers and consumers. Figure 3.3 shows which application uses which routing keys and Figure 3.4 shows the corresponding queues and which applications consume from these queues.

Routing Key	Purpose	Publisher	Consumer
register_confirmation_email	Sending a confirmation email	Core	Email
register_summary_email	Sending a summary email	Core	Email
transaction_tokens_allocated_email	Sending an email when tokens are allocated for a specific transaction after a detected transaction reaches a previously specified number of confirmations	Monitor	Email
transaction_received_email	Sending an email as soon as a transaction is detected	Monitor	Email
kyc_start_email	Sending an email when the KYC process with a specific investor is started	KYC	Email
kyc_reminder_email	Sending an email when an investor is reminded to complete a previously started KYC process	KYC	Email
address_set_wallet	Adds the provided addresses to the monitor module to be monitored	Core	Monitor
block_nr_bitcoin	Sending the currently downloaded highest Bitcoin block number	Monitor	Rates
block_nr_ethereum	Sending the currently downloaded highest Ethereum block number	Monitor	Rates
exchange_rate_request	Requesting the USD per cryptocurrency unit exchange rate at a specific timestamp. This is the only queue on which the sender awaits a response from the consumer.	Monitor	Rates
kyc_start_email_sent	Confirming that KYC start email was sent successfully	Email	KYC
kyc_reminder_email_sent	Confirming that KYC reminder email was sent successfully	Email	KYC

Table 3.1: Overview of all routing keys used by ICONator.

Since every communication type uses a separate routing key and queue, the messaging system in ICONator is easily extensible. The routing key and queue names (constants) and the corresponding message objects are defined in the `commons` module `amqp-model`. For creating a new message types, one can add a new message class and the corresponding queue and routing key constants. Those can then be used to publish a message to the new queue or consume a message from the new queue.

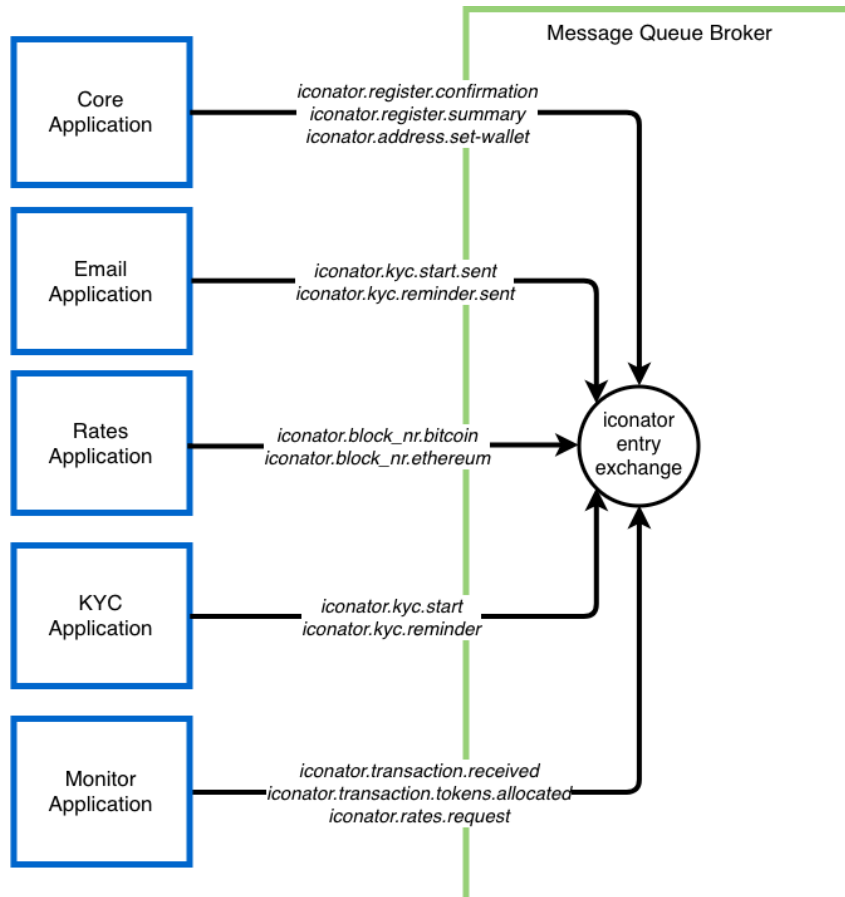


Figure 3.3: Routing keys used by each application to send messages to the exchange.

3.3 Source Code Structure

As the ICONator consists of multiple decoupled modules, the source code is structured accordingly. The five applications are gathered within the `services` package. Every application is a separate Spring Boot application and can be started on its own. Parts of ICONator that are used by multiple applications from the `services` package are put in their own module within the `commons` package. This avoids duplicate code and leads to better maintainability. Which `commons` modules are used by which application can be seen in the `dependencies` scope of the respective `build.gradle` files.

Every module in ICONator comes with its own unit tests. Those tests can be run all at once or independently of each other. The unit tests reside in the `test` folder

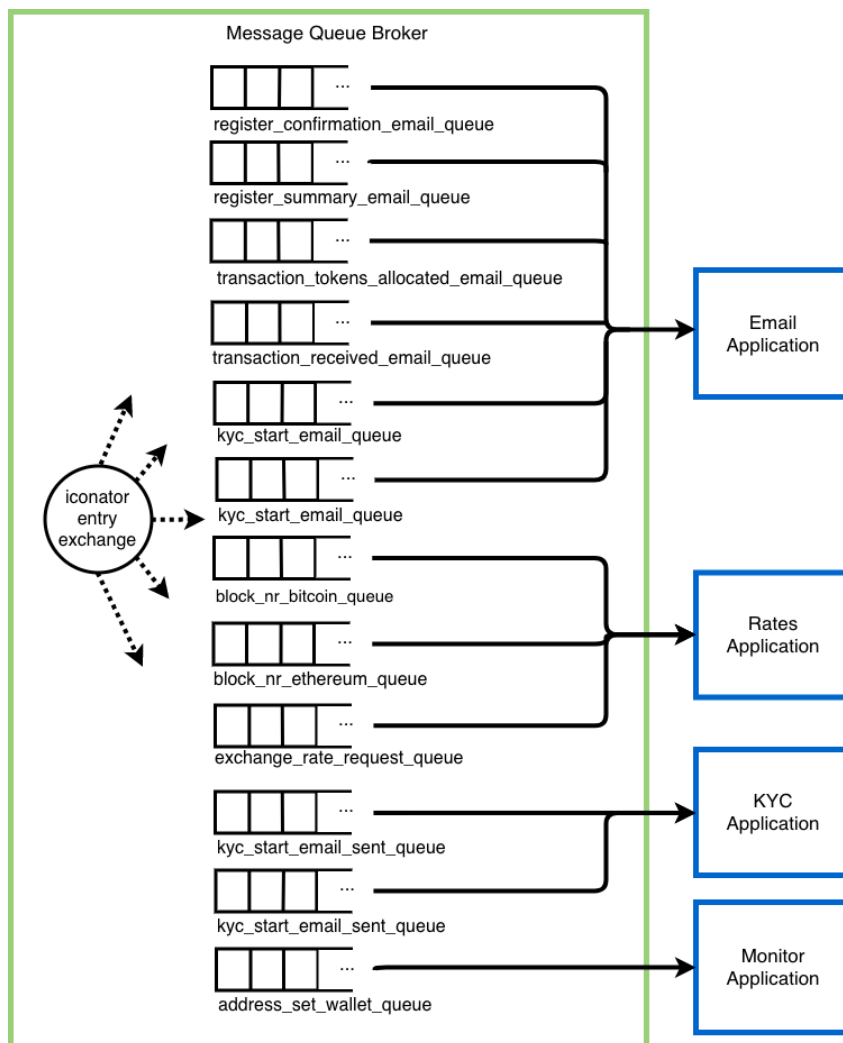


Figure 3.4: Message queues used by each application to consume messages.

of each module and follow the same package structure as the source code (e.g. the test for the `EthereumMonitor` in `services/monitor/src/main/java/io/iconator/monitor/EthereumMonitor` is found in `services/monitor/src/test/java/io/iconator/monitor/EthereumMonitorTest`).

The following shortly describes the contents of the `commons` modules. The modules in `services` are not mentioned because they each have a dedicated section in Chapter 4.

commons.amqp-service Contains the service which provides all the message sending methods used by the ICONator's applications to exchange information over the message broker.

commons.amqp-config Contains a facade for AMQP operations (send and receive) and an asynchronous implementation of it. These classes provide abstraction for specific AMPQ client libraries (like RabbitMQ). They are used by the `commons.amqp-service` module.

- commons.amqp-model** Contains all message classes used to exchange information between the IConator's applications. Contains definitions of string constants for queues and routing keys.
- commons.bitcoin** Contains Bitcoin-related enum definitions, service, utility and configuration classes. Includes the bitcoinj library.
- commons.country-filter-service** Contains a service for assessing if an IP address should be allowed or not. Contains country filter related configuration classes. The country filtering is in the core module when getting registration requests from investors.
- commons.cryptocompare-service** Contains a service and required DTO classes for accessing historical exchange rate data from the CryptoCompare API. Contains CryptoCompare-related configuration classes and enum definitions. The data from CryptoCompare is consulted by the rates application if there is a gap in the exchange rates database in which exchanges rates have not been recorded properly.
- commons.db-services** Contains services that provide database access and other common methods concerned with the IConator's entities .
- commons.ethereum** Contains Ethereum-related enum definitions, service and utility classes. Includes the ethereumj and web3j libraries.
- commons.mail-service** Contains all email sending-related services and configurations. It also contains all the functionality for building the email content
- commons.model** Contains the entity models and some common enum definitions.
- commons.recaptcha-service** Contains a service and corresponding configuration for captcha verification.
- commons.secutiry-actuator** Contains the configuration concerning web security.
- commons.services-base** Contains other code useful to the **services** modules.
- commons.sql-repository** Contains all database repositories. Repositories are a Spring concept and encapsulate the accesses to the database. Other modules should not access the repositories directly but rather use the database services in **commons.db-services**.
- commons.test-utils** Contains test utilities used in unit testing.

Chapter 4

Applications

This chapter contains detailed information about the modules which make up the ICOinator platform. Each section concentrates on one of the modules and describes its functionality and the API endpoints offered. Each module is its own Spring Boot application and can be run as an independent application, although there exist dependencies at runtime which are illustrated in Section 3.2 and are mentioned in each application's Section.

4.1 Core

While the core application offers various functionality, it is mainly concerned with the registration of investors which want to take part in the ICO and need to get payment addresses to which they can pay to. The functionality is split up in multiple controllers. The API endpoints that each controller in the core module offers are described in Section 4.1.2. The aforementioned register functionality is found in the **RegisterController**, while setting the addresses is done via the **AddressController**. Section 4.1.1 shows the steps involved in completing an investor registration for an ICO and what is happening internally. As additional functionality, the core application also offers the import of key pairs, an email whitelist and returns information about the tier setup and the countries that are allowed to participate in the ICO.

4.1.1 Registration Steps

The following steps describe the actions that are needed in order to register an investor for an ICO.

1. Investor begins the registration with a POST request at the **/register** endpoint containing only the email address and an optional ReCaptcha response token.
2. Investor receives an email with a confirmation token for his registration.

3. Investor validates the account using the provided link in the email. This calls the `/register/<emailConfirmationToken>/validate` endpoint.
4. The addresses for the investor need to be set with a POST request to the `/address` endpoint with the addresses to set and the confirmation token as authorization.

After the last step, the account for the investor is set up. From then on, the investor's payments get tracked and notification emails get sent out.

4.1.2 Core API

AddressController

`/address`

- Setting the wallet address of investor and the refund addresses in Ethereum and Bitcoin (ETH, BTC), setting and distributing the pay-in addresses. The authorization header uses the confirmation token as authorization.
- Method: POST
- Data Params:
 - Required: AddressRequest
 - Required: authorizationHeader

CountryFilterController

`/countryfilter/allowed`

- Returns `true` or `false` if the country filter is enabled or not.
- Method: GET

KeyPairsController

`/keypairs/import`

- Imports the keypairs from a CSV file and returns the amount of successes and failures.
- Method: POST
- Data Params:
 - Required: inputStream

RegisterController**/register**

- Registers a new user. If the investor has a wallet, and all the payIn addresses set, then, it sends the summary email. Else, it sends the confirmation email with the confirmation email token.
- Method: POST
- Data Params:
 - Optional: ReCaptcha response token
 - Required: RegisterRequest

/register/<emailConfirmationToken>/validate

- Validates the email confirmation token.
- Method: GET
- URL Params:
 - Required: emailConfirmationToken

TierController**/tiers**

- Returns the ICO tiers in a Json format.
- Method: GET

/tiers/create

- Creates the tiers, perfomed once in the beginning of the ICO.
- Method: POST
- Data Params:
 - Required: tiersRequest

WhitelistEmailController

/whitelist/subscribe

- Subscribes an email address to the whitelist and confirms the subscription.
- Method: POST
- Data Params:
 - Required: `whitelistEmailRequest`

/whitelist/emails

- Returns all the whitelist emails.
- Method: GET

4.2 Email

The email module takes the responsibility for handling all emails that should be sent from within IConator. It is fully decoupled from the rest of the modules and does not interact with any database. Every IConator email needs to correspond to a pre-defined email type. Such an email type consists of an HTML template that is fully customizable. The templates are explained further in Subsection 4.2.2.

4.2.1 Email implementation

In order to achieve full independence from the rest of the applications, the email module only consists out of message consumers. There is a message consumer for every type of email that IConator is able to send. This means that for every new type of email that should be sent by IConator, a new template needs to be created. The actual building and sending of the email is then delegated to a different `commons` module.

Figure 4.1 describes how the email module works in detail. If any application in IConator needs to send an email, a message corresponding to the email type needs to be queued into the email message queue. The process to send the email is then started as soon as one of the message consumers in the email module picks up a message requesting an email to be sent. The email message consumer from the email module calls the corresponding `sendMail` method from the `commons` module `mail-service`. The `mail-service` module first builds the mail. In order to build the email, `mail-service` calls the `MailContent-Builder` which uses the Thymeleaf template engine to create the email out of the prepared template. As soon as the email is built successfully, `MailService` then delegates the task of sending the email to the external SMTP server. The SMTP server picks up the email and finally delivers it to the recipient.

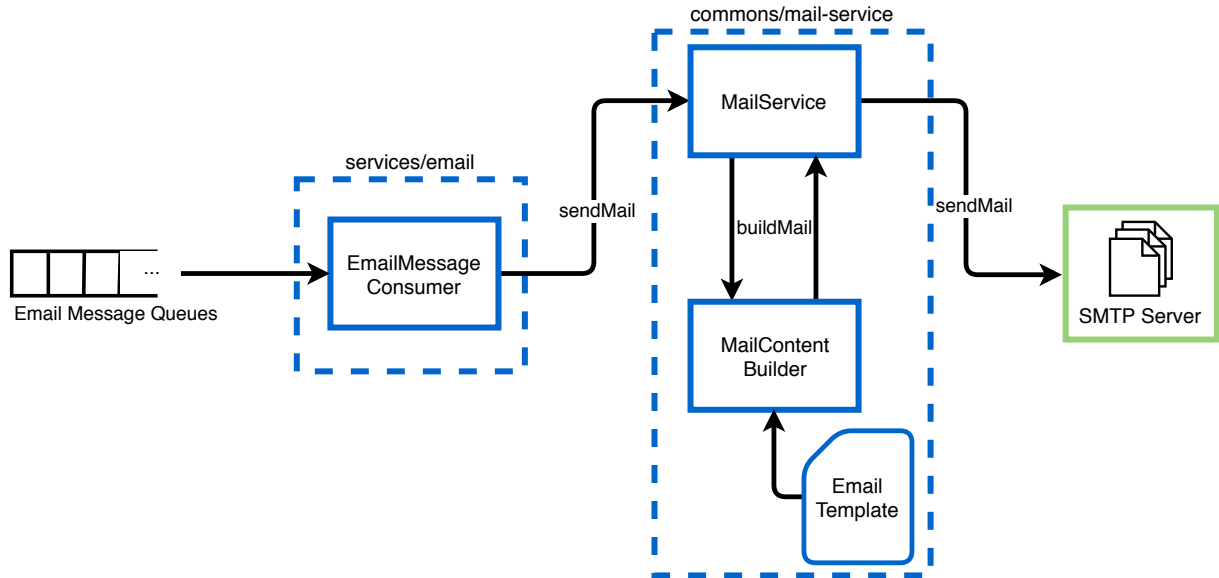


Figure 4.1: Diagram of the email module.

4.2.2 Email templates

Every type of email needs its own HTML template. The templates are stored in the `ressources` folder of the `commons` module `mail-service`. The templates are interpreted by the *Thymeleaf* template engine which allows for embedding Java variables within the HTML [32]. All templates contain placeholder variables for defining the name of the token sale, the name of issuing entity and the year. Those variables can be defined via the application properties.

Confirmation email

The confirmation email is always the first email received by an investor. The email gets sent after signing up for the ICO. It contains a generated token with which the registration needs to be confirmed before further actions can be taken. This enforces users to provide a valid email address, as otherwise the token cannot be confirmed.

Transaction received email

The transaction received email is sent as soon as a transaction on either the Bitcoin or the Ethereum blockchain is detected. The email serves as a first information for the investor while ICONator is waiting for enough confirmations on the detected transaction. It contains the amount of funds detected in the transaction and a link to track the transaction.

Tokens allocated email

The tokens allocated email is sent after a previously detected transaction has received enough confirmations. The email contains the amount of funds paid in and the corresponding amount of tokens that were bought with the payment.

Summary email

The summary email contains a summary of all the account details. It serves as an information for the investor. The summary email lists the Bitcoin and Ethereum pay-in addresses, the token wallet address and the refund addresses that were added by the investor during the registration process.

KYC start email

The KYC start email is sent to an investor when the KYC process is started by the KYC module (see Section 4.5). The email contains a link to a website where the KYC process should start. This link can either be supplied manually or generated automatically by the KYC module.

KYC reminder email

The KYC reminder email serves as a reminder to the investor for completing the KYC process. The email contains the same link as in the KYC start email.

4.2.3 SMTP server

ICOnator emails are sent via an external SMTP server. The SMTP server must be set up independently of ICOnator in order to send emails successfully. ICOnator supports any generic SMTP server. The configuration variables ICOnator needs to access the SMTP server are found in the application properties of the email module and are further described in Section 5.3.1.

4.3 Monitor

This section contains an overview of the functionalities of the monitor application and discusses important design and implementation decisions made.

Section 4.3.1 contains a brief explanation of what the monitor application is and does. It also states important properties of the monitor.

Section 4.3.2 introduces the concept of sale tiers on which the monitor relies on when allocating tokens.

Section 4.3.3 describes the behavior of the monitor application in regard to concurrently running instances and unexpected crashes. It discusses considerations about scalability and robustness.

Section 4.3.4 briefly explains the behavior of the monitor when new investors register while the monitor is already running.

Section 4.3.5 describes the concept of transactions and their attributes in context of the monitor application.

Section 4.3.6 describes how transactions are processed leading to token allocation or the creation of refund entries.

Section 4.3.7 discusses how monetary values are modeled and handled within the monitor application.

The terms *transaction* and *payment* are frequently used in this chapter. They both refer to a payment made via a blockchain network and are used interchangeably.

4.3.1 Overview

The task of the monitor application is to monitor blockchain networks and process incoming transactions made by investors. Only the monitor application interacts with the blockchains, none of the other IConator modules do. It allocates tokens according to the incoming payment amounts and informs the investors about their investment. The monitor application interfaces with blockchain networks via libraries like bitcoinj [33] for the Bitcoin blockchain and web3j [34] for the Ethereum blockchain. If a transaction is directed to a monitored payment address, that transaction is picked up and processed. The transaction processing includes sending confirmation emails to the investors who made the payments, allocating tokens or creating a refund entry if the processing runs into an error. The diagram in Figure 4.2 visualizes the described activities on a high abstraction level. The application supports the Bitcoin and Ethereum network. Via the configuration one can define to which networks an instance of the monitor application should connect to. Thereby, an operator of the IConator could run one monitor instance monitoring both blockchains, or he can run two instances each monitoring one of the blockchains. Running even more instances is possible, but the two cases above are the most plausible ones.

Because the monitor handles monetary transactions, it is of high importance that it is robust to failures. That means that in the end an investor must either be allocated the correct amount of tokens according to his payment or a refund entry must exist that entitles him to get his payment refunded. On the other hand, in the interest of the operator of the IConator, the allocated tokens must exactly match the received monetary value.

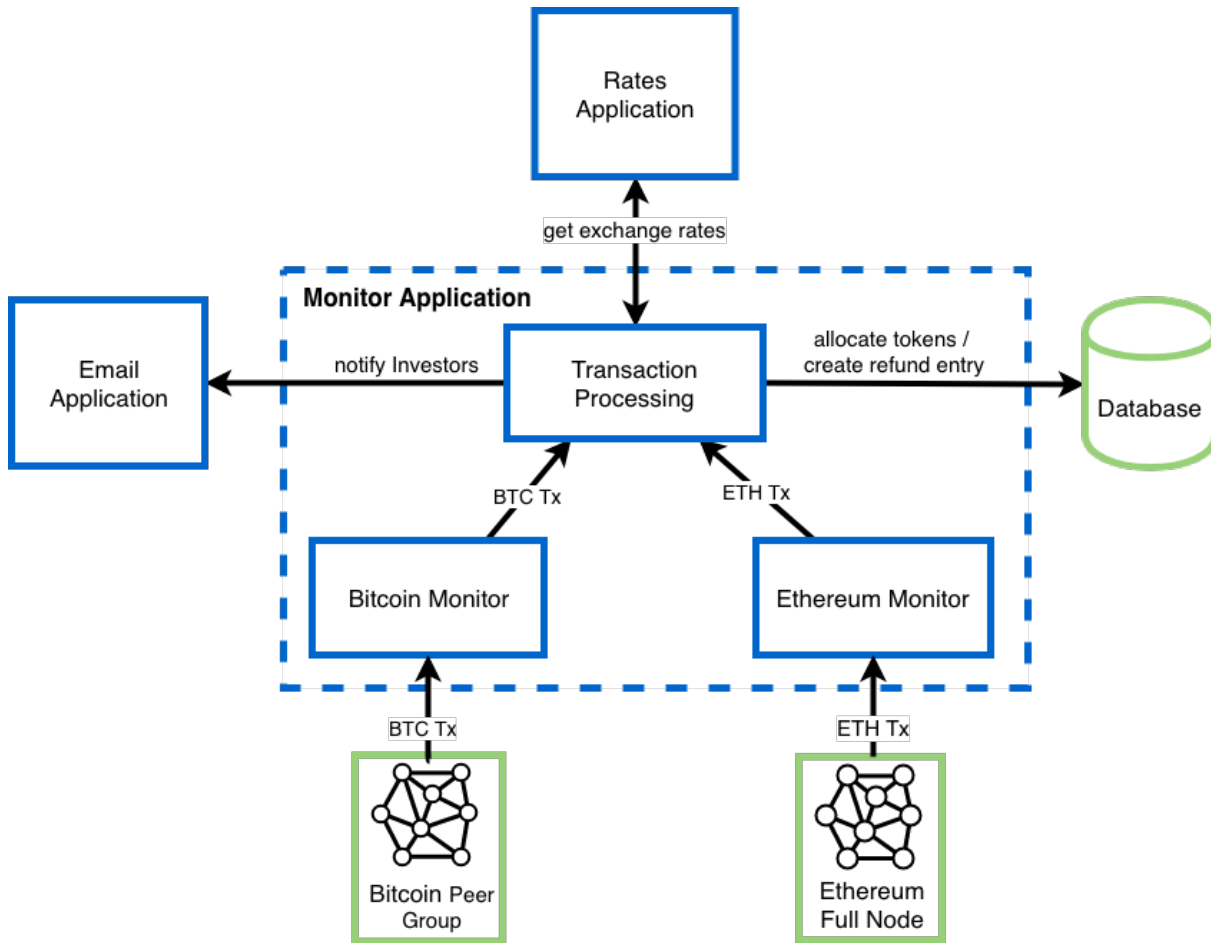


Figure 4.2: Diagram of the monitor application and its surrounding systems.

4.3.2 Sale Tiers

The concept of sale tiers is commonly seen in ICOs. The Tanzo [35] or nOS [36] tokens are mentioned here as examples. Tiers divide an ICO into multiple parts. Thereby, the total token limit of the ICO is divided into buckets. Only one tier is active at a given time and an investment made at that time will take tokens from the corresponding active tier if possible.

Throughout the report and the ICONator's source code the meaning of "taking tokens from a tier" can also be expressed as "distributing a payment to a tier". Similarly, when talking about a tier's token reservoir being depleted, the opposite expression of the tier being filled to its token limit is used sometimes.

The ICONator implements the concept of tiers in the **SaleTier** entity and provides several attributes of which the following are commonly seen.

Token limit the maximum amount of tokens that can be sold from the tier.

Tokens sold the tokens sold from the tier.

Date range the date range in which the tier is active.

Discount the discount applied to tokens sold from the tier.

Additionally, the IConator differentiates between static and dynamic duration and static and dynamic token limit. These two attributes required further explanation and are discussed in the following.

Sale Tier Duration

Tiers can behave in two ways in respect to their date range.

Dynamic duration tiers change their end date and the dates of all successive tiers when their token limit is reached and no more tokens can be sold from them. The tier's end date is shifted to the moment at which the token limit is reached and the start and end dates of all successive tiers are shifted by the same amount of time without changing their duration. After the dates have been adapted, the subsequent tier is now the active one and new payments are directed to it.

Static duration tiers do not adapt any dates when their token limit is reached. Further payments to such a tier, i.e. payments made in its active period, are automatically distributed to the subsequent tier without shifting the start date of that subsequent tier to the current moment.

This implementation leads to more than just an active/inactive status of sale tiers. Tiers can rather be in status formed by any combination of active/inactive and open/closed. For example, a tier which is not active yet because its start date is in the future is in status inactive and open. It is open because it has not reached its token limit yet. Payments will be distributed to it as soon as its preceding tier is full even if its start date is still in the future. Distributing payments to inactive tiers might seem unreasonable. But without this any payment made in a period in which no tier is active would result in a refund entry. These refunds are an unnecessary burden on the operator of the ICO. The rationale behind the design is that if an investor makes a payment in a period where no tier is shown as open and active, it is assumed that the payment should be distributed to the tier that will become active next.

Sale Tier Token Limit

Tiers can behave in the following ways in regard to their token limit:

Dynamic limit tiers have no initially set token limit. The token limit is set as soon as they become active. It is calculated by subtracting the so far sold token amount from the overall token limit of the ICO.

Static limit tiers have a fixed token limit configured at the beginning of the ICO. It doesn't change when they become active.

Configuring the sale tiers

To set up a constellation of sale tiers in the IConator, one can either make calls to the REST endpoint `/tiers/create` provided by the core application or run a script on the database directly. In both cases the same information has to be provided for setting up a tier.

Tier number The numbering of the sale tiers is used to determine their ordering. One has to make sure that the numbers reflect the dates ordering and are gapless. This is also the unique identifier of the `SaleTier` database entity.

Description A short description of the tier.

Start date The date and time at which the tier should become active. If only a date string is provided, e.g. `'2018-10-09'`, the start time is at midnight at the beginning of that day. This attribute might get changed by the program logic during the ICO because of dynamic duration tiers.

End date The date and time at which the tier should become inactive. When only a date string is provided, e.g. `'2018-10-09'`, the end time is at midnight at the beginning of that day. I.e. the whole day specified is not part of the tier's date range. This attribute might get changed by the program logic during the ICO because of dynamic duration tiers.

Discount The discount applied to the token price in this tier. The value must lie between 0 and 1. Exclusive 1 because it is not possible to give away tokens for free with the IConator. For example, if the discount is set to 0.2, the tokens will be sold at 80% of their actual price.

Tomics sold The amount of tokens in their atomic unit sold so far from this tier. Usually, when setting up the tiers, this value would be set to 0. Since the value is stored in the atomic unit of the tokens, fractional values are not possible.

Tomics max Specifies the tokens limit in atomic units for this tier. When this amount is reached in the sale, the tier becomes inactive. Since the value is stored in the atomic unit of the tokens, fractional values are not possible. **Important:** Set this to 0 or `null` if the tier has a dynamic token limit.

Has dynamic duration Specifies if this tier has a dynamic duration that adapts as soon as its token limit is reached and changes all the following tiers' start and end dates.

Has dynamic max Specifies if this tier has a dynamic token limit, which would make his token limit dependent on the overall available tokens in the sale at the time the tier becomes active.

See Appendix B for an example of a sale tier configuration given in an SQL script that could be executed on the IConator's database. A request to the REST endpoint `/tiers/create` could be set up similarly.

4.3.3 Monitor instances and concurrency

At start up of an application instance it is determined which blockchains should be monitored. This can be specified in the corresponding configuration variables. The pay-in addresses corresponding to the configured blockchain are then read from the database and put into a simple list or a wallet implementation. These are the addresses that have been distributed to the investors when they registered for the ICO. For example, if a monitor instance is configured to only monitor the Bitcoin blockchain, then only Bitcoin addresses will be loaded and monitored. Every started instance loads all distributed pay-in addresses for the blockchains it has been configured for. It is not possible to have multiple instances work on those addresses in a chunked and separated way. If two instances monitor the same blockchain they will both try to process the same transactions. But the implementation makes sure that no transaction is processed twice. A monitor instance will drop a transaction if it sees that another instance is already processing it or has processed it. This induces a minor load balancing between the two instance. But we assume that it does not lead to much horizontal scalability because every monitor instance still has to take at least look at every transaction, even if it does not process it. We have not made any tests to confirm or disprove this.

More scalability could be provided by placing a separate load balancer in front of multiple monitor instances that distributes incoming transactions evenly. Each monitor only sees a chunk of all transactions. Another way of dividing the work between multiple monitor instance would be by splitting the monitored addresses among the instances. The following paragraph shows the reasoning why we decided against this.

By assigning each instance a chunk of the addresses, each instance will still have to scan through all blocks at start up but only process the transactions that correspond to their monitored pay-in addresses. This approach obviously demands that the addresses in the database and new addresses received via the message queue are split up reasonably between the instances. This poses a first problem. The monitor instances are separately running applications that have to negotiate through message queues or the shared database who will handle which addresses. Assuming that the splitting up works and multiple instances currently monitor their addresses. If one instance crashes a second problem arises. The addresses that it previously monitored become unmonitored and a solution has to be found for how those addresses should be either assigned to still running instances or be picked up again by a restarting instance.

Implementing solutions for the posed problems above seemed to introduce too much complexity into the logic of the application. It was decided to not add the possibility of splitting up the pay-in addresses between monitor instances.

The relative¹ low number of transactions per second on the Bitcoin and Ethereum networks at the time of writing do not demand much scalability of the monitor. Basic scalability is provided by running separate monitor instances per blockchain. Of course if

¹ Relative to the transaction volume that for example the Visa network can handle. Visa claims to be able to handle 65000 transaction per second [37]. Bitcoin with about 7 and Ethereum with about 25 transactions per second are yet to be scaled to high transaction volumes [38, 39].

in the future a blockchain network which much higher throughput should be monitored the monitor application might need a rework to provide for splitting of monitoring work.

Besides the scalability considerations robustness is also an important property of the IConator. Running more than one monitor instance concurrently, where all monitor the same addresses, brings more robustness into the transaction processing. If one instance crashes the other one will go on monitoring unhindered and transactions will be processed as fast as before the crash.

But there are reasons why a crashed monitor instance should be restarted. If the instance crashed while processing a transaction, it will leave it in an incomplete state. If at the same time the still running instance has already dropped the transaction because it has seen that another instance was processing it, the transaction will not be finalized. Instead, a restart of the crashed instance is necessary to make sure that the incomplete transaction is picked up again and finalized. The restarting instance does not particularly remember the transaction but will see it when catching up to the newest block.

4.3.4 Adding monitored addresses

While the monitor instance is running, messages of newly registered investors arrive via a message queue. The investor's payment addresses are then added to the monitored addresses and equally treated as the ones loaded from the database at startup. This implies that if a transaction is made in the small time span between the investor registering and the message of the new investor reaching the monitor then the transaction will go unnoticed. Though it is very unlikely that this will actually happen. And even if it should happen then the transaction could be picked up again by restarting the monitor because it will then receive the investor's payment addresses directly from the database.

If multiple monitor instances monitor the same blockchain each of them will get every new investor messages since each of them listens to an own queue that gets the investor messages from the message exchange. In this way no monitor instance is the only one responsible for a transaction and if an instance fails other instances are still monitoring the same transactions as described in Section 4.3.3.

The exchange and the queues which direct the new investor messages to the monitor are set to be durable to ensure that they are set up again if the message broker has to be restarted. The investor messages are not persistent by default, which means that they could be lost when the message broker suddenly stops running. However, this can be configured in the message queue broker. Or if in doubt a monitor instance could be restarted to load the missed new investors from the database.

4.3.5 Transactions

Transactions are abstracted by the `TransactionAdapter` interface. For every blockchain that should be monitored an implementation of that interface, besides other things, must

be created. The **TransactionAdapters** are used to abstract away the specifics of different blockchain interfaces and unify the processing of transactions independent of the blockchain it arose from. Objects of this type are not stored in the database.

The interface expects that a blockchain transaction can provide the following information:

- a transaction identifier
- a transaction value which is the payment amount
- the address of the receiving account
- the block number and the block time in milliseconds of the transaction's block

These do not cover all of the attributes/methods of the interface but are the ones which have to be retrieved from the corresponding blockchain and can probably not be easily derived in other ways.

What is not represented in **TransactionAdapter** is a transaction status. The status is the confidence into a transaction, i.e. how probable it is that the transaction will stay in the chain. Handling the status is up to the blockchain specific implementation of the **BaseMonitor** class. The monitor application differentiates between three different status in which a payment can be in.

- A payment which was just discovered on the network but does not yet sit on a block is in status *pending*.
- A payment which has been added to a block is in status *building*.
- And a payment which is covered by a certain number of block is in status confirmed. The number of blocks which are necessary for this status is configurable.

The monitor application, more precisely the blockchain specific implementations, react to all three of these statuses. Transactions which are *pending* are processed simply to inform investors about the arrival of their payment as soon it is seen on the blockchain network. Once the transaction reaches the *building* status, it is processed for token allocation. After the token allocation the monitor keeps on tracking the transaction until the configured confidence level is reached and the status can be set to *confirmed*. The last step, tracking the transaction after processing and setting the status to confirmed, is not unified the **BaseMonitor** while the other two processing steps are. Therefore, it is the task of the specific monitor implementations to handle the confirmation of transactions individually.

4.3.6 Transaction processing

This section leads through the procedure of transaction processing and discusses decisions and considerations that have been made in development.

When the monitor application is started it begins to catch up to the newest block on the blockchains it was configured for. In the case of the **EthereumMonitor** the block at which it should start can be configured so it is not necessary to go through the whole block history. In case of the **BitcoinMonitor** the block at which the monitor starts catching up depends on the specified time stamps when adding monitored addresses to the bitcoinj wallet. These time stamps cannot be configured they are simply the date on which the associated investor was registered. Because of the way the transaction processing works it is important that after a unexpected crash the blockchains are scanned again starting at the same blocks as were initially set. This guarantees that all transactions which were partially processed or missed because of the crash will be reprocessed correctly.

Two processing flows

As described in Section 4.3.5 the monitor application differentiates between pending and building transactions. The **BaseMonitor** implements two corresponding separate processing flows. Blockchain-specific implementations of the **BaseMonitor** can therefore have a transaction be processed depending on its status by calling the corresponding method of the **BaseMonitor** and providing the transaction object. The two existing implementations the **BitcoinMonitor** and the **EthereumMonitor** are implemented like that.

It is not absolutely necessary to process a pending transaction since the truly relevant things happen in the processing of building transactions. The benefit of it is that the investor is informed about the arrival of his payment as soon as the transaction is seen in the network.

The two processing flows can be viewed in the flowchart in Appendix C. The draw.io [40] source file for the diagrams (for more comfortable viewing) can be found on the IConator repository [41]. The numerous checks that can be seen in the flowchart which are also present in the source code are a result of the concurrency mentioned in Section 4.3.3. They ensure that tokens are never allocated more than once for a transaction and that a monitor instance can tell if a transaction is in a unfinished state, e.g. after another instance crashed while processing it.

Payment logs

The transaction processing uses the **PaymentLog** entity to save all relevant information about the processed transaction. A monitor instance commits a payment log to the database as soon as it starts processing a transaction (pending or building). This allows multiple monitor instances to be synchronized and informed about the work of the other instances. Since there are two separate processing flows for the two status pending and

building a transaction may be processed by two different monitor instances. One when it's in the pending and the other when it's in the building status. But the payment log that the two instances add information to is the same one.

Refund entries

Apart from the `PaymentLog` there is the `EligibleForRefund` entity which is also important here. If anything fails on the way to allocating tokens for a payment then the monitor creates a refund entry which states why the processing failed and that the investor should get a refund. Reasons for this may be:

- An essential transaction information cannot be retrieved from the blockchain.
- The exchange rate cannot be fetched.
- The payment amount is not sufficiently high.
- A part or all of the payment cannot be allocated in tokens because all sale tiers are already full or the overall token limit has been reached.

If a payment log is incomplete, e.g. because no tokens have been assigned, but a refund entry for the transaction is present the transaction is still interpreted as fully processed.

Transaction confirmation

After a transaction has been processed either successful or by creating a refund entry the transaction is further tracked for confirmation. This part is not handled in the `BaseMonitor` but by the blockchain-specific implementations. The `BitcoinMonitor` and `EthereumMonitor` don't handle this exactly the same because the capabilities of the bitcoinj and web3j libraries differ. But in both cases the tracking for confirmation works with multiple monitor instances and is robust against crashes. In case of a crash of a monitor instance the transaction will be picked up again after a restart of the instance. This works because on startup the instance will catch up to the newest block starting from the configurable starting block and will thereby continue the tracking for confirmation for transactions that were not yet confirmed. In case of multiple instances running and one crashes, the still running instances will manage the tracking for confirmation even if they weren't the ones that processed the transaction.

Further considerations

In the following more details and considerations about the transaction processing are discussed in a question-answer manner.

What if a monitor instance starts processing a transaction, creates a payment log and stops unexpectedly without a graceful exit before being able to finish the processing?

It can happen that a monitor creates a payment log, then starts processing the transaction but stops unexpectedly without a graceful exit. What remains is a transaction that was not fully processed and the corresponding payment log, possibly with incomplete information. The transaction needs to be picked up by a restarted monitor instance to continue the processing. It is unlikely that a already running instance will pick up the same transaction after the first instance failed to process it because they both receive the transaction at about the same time from the blockchain. The second instance recognizes the transaction checks if there is a corresponding payment log and sees the one that the first instance already created. Then at the time that the first instance fails and stops the second instance already moved on to other transactions. Therefore, it is inevitable that the incompletely processed transaction needs to be handled by a restart of the failed instance.

To determine if an unfinished payment log is still being processed or not, the monitor simply checks when the payment log was last modified. If that date-time is in the past by a specific, configurable time interval it is assumed that the monitor instance that processed the transaction crashed and left it in an unfinished state. This time needs to be long enough to permit full processing of a transaction and needs to be shorter than the time needed to detect a crash and restart an instance. The restart scenario should look like the following. A monitor instance crashes and is restarted. As soon as it is up and running again it catches up to the newest block and thereby reaches the incompletely processed transaction. The date of last modification of the corresponding payment log should now be at least the configured time interval in the past. With that information and seeing that the payment log is incomplete the monitor can reprocess the transaction.

Before reprocessing of a transaction is considered by a monitor instance, it checks the payment log's attributes that give clues about its completeness. Payment logs are always created with the transaction status attribute set. Therefore, it is always clear in which status the transaction was handled last. In each status, pending or building, there are different criteria for what makes the transaction fully processed. If a pending transaction is fully processed the `is_transaction_received_message_sent` attribute on its payment log will be set to true. If a building transaction is fully processed the `is_allocation_message_sent` attribute of its payment log will be set to true. Other attributes on the payment log are set along the way of processing and are used to determine the degree to which the transaction was processed. For example if the processing has reached and finished token allocation but then abruptly stopped without sending the information email to the investor it is important that the token allocation is not repeated by the next monitor instance picking up the incompletely processed transaction. It only has to send the information email. The transaction processing is therefore split into a several steps where each step happens as a database transaction to guarantee that either all operations in that step are persisted or non of them. For example the token allocation and updating of the payment log with the allocated token amount have to happen in the same transaction otherwise an unexpected failure could result in an inconsistent state of the data.

Why don't we use database locks on the payment logs?

Locking a payment log as long as processing its transaction would have been another way of

synchronizing multiple monitor instances. This would make it clear that a transaction and its payment log are currently processed because of the locked column on the database. But in case of a crash of the monitor instance the lock might remain for an unknown amount of time because the database server did not get any specific interrupt signal for canceling the ongoing transaction holding the lock. Therefore, a restarted monitor instance will not touch the incompletely processed transaction because it sees that it is still locked. Another problem with this approach is that if a transaction is picked up in status building but is concurrently also being processed in status pending the building part would have to wait until the payment log is unlocked which is not necessary. The two processing flows do not hinder each other. With the chosen implementation this concurrent processing in both status is possible.

What if a transaction was never seen in status pending but immediately went to status building?

Such transactions are simply processed in status building directly. This implies that the first confirmation email is not sent to the investor. Though the token allocation email will be sent nonetheless after the processing in status building is completed successfully.

What if a transaction was successfully processed in status building but then is invalidated on the blockchain and not added again?

Such transactions are not reprocessed. The monitor only observes status changes from pending to building to confirmed. If after an ICO a payment log is not in the confirmed status it means that the validity of the payment must be checked manually.

4.3.7 Modeling of currencies and tokens

When working with monetary values one has to make sure that the used data type representing the value can give exact control over the value's precision and that one does not introduce cumulative rounding errors when manipulating the value. The monitor is the only application in which monetary values are not only read but also manipulated and stored. The rates application calculates values of exchange rates and therefore needs a defined understanding of value precision too but it doesn't need to do currency conversions calculations as the monitor does.

Monetary values are either fiat currency, cryptocurrency or token values.

The ICONator uses Java `BigDecimal`s with a precision of 34 with the half even rounding mode to manipulate monetary decimals. This corresponds to the `Decimal128` floating-point format which is part of the IEEE 754 Standard. Therefore, a monetary decimal can at anytime have at most 34 significant digits. The scale of a `BigDecimal` that is being processed is not explicitly set but chosen by the Java run time environment.

Fiat currency values are processed as just described. Only when they are stored in the database the scale is explicitly fixed to 6, i.e. they have 6 digits in the fractional part. This implies that a stored fiat currency number can at most have a magnitude of $34 - 6 = 10^{28}$ which is surely large enough to handle USD amounts. If nevertheless a value with higher

magnitude is stored the database system will probably return an error. If the scale of a value to be stored is greater than the declared scale on the database attribute, the database system will round the value to the specified scale. Usually a simply rounding down mode will be used.

Cryptocurrency and token amounts are only held in the Decimal128 floating-point format when doing calculations (e.g. conversion from fiat to cryptocurrency). As soon as a final result has been computed the decimal is cast to an integer number. This is done because it is assumed that cryptocurrencies and tokens can be represented in an atomic unit (e.g. Wei for Ether) that cannot be fractional. This is why they are represented by `BigIntegers`. The decimal value in the atomic unit is cast to the integer simply by removing the fractional part. I.e. any fraction of the atomic unit is removed which is equal to rounding down. When saving cryptocurrency or tokens amounts to the database the precision/number of significant digits is set to 34. In the example of Ether this gives the possibility of saving Ether amounts up to the magnitude of 10^{16} which is big enough given the value of the ether at the time of writing (ca. 180 USD per one Ether).

Exchange rates are stored and used as decimals with a scale of 6. With that we can cover conversions of amounts up to one million with accuracy to the first digit before the decimal point. For example 1'000'000 USD times the exchange rate 0.3005064 (scale of 7) would result in 300'506.4 but with the chosen scale of 6 the 40 cents are lost, and we get only 300'506 USD. But a higher scale doesn't make sense because exchange rates retrieved from exchange rate services usually don't provide more digits after the decimal point.

4.4 Rates

This section describes the rates module. Things that are discussed are how exchange rates are fetched, what fallback mechanisms are in place if the main sources of rates are not available, how the rates application interacts with the monitor module and how its web API looks like.

The rates module has the main purpose of continuously storing exchange rates of different cryptocurrencies and making them available for other applications. The rationale behind having an own provider of exchange rates is simply self-sufficiency and more control over availability.

4.4.1 Fetching rates

The cryptocurrencies for which exchange rates shall be fetched as well as the base fiat currency can be configured. The rates are fetched via a task that is run repeatedly in a configurable time interval. Each time the task is run the exchange rates are fetched from different exchanges which can also be configured. The list is constrained by the exchange names given in the `ExchangeType` enum which map to corresponding exchange-interfacing implementations in the XChange Java library [42]. The rates collected from the different sources are cleaned from outliers and averaged. If non of the configured exchanges can be

reached no aggregated exchange rate entry is created for that particular run of the rates task. I.e. there will be a gap in the rates history. Section 4.4.1 provides information on how gaps are handled.

The information that is stored when fetching exchange rates is contained in four entities. First for every configured exchange service a **ExchangeEntryRate** is created and several **ExchangeCurrencyRates** are added to it each corresponding to a currency type. Then an **ExchangeAggregateRate** is created and populated with multiple **ExchangeAggregateCurrencyRates** of which each is an aggregate of the **ExchangeCurrencyRates** from before, corresponding to the same currency type.

Next to the creation date, currency type, exchange rate, and the exchange service, from which the rate entry has been fetched, we also store the block number which was current at the time that the rate was fetched. The implementation stores the block numbers for Ethereum and Bitcoin in the corresponding attributes of the **ExchangeAggregateRate**. The block number only has logging purposes and is not used anywhere else in the IConator implementation. The source of the block number is the monitor application because it is the only part of the IConator which interfaces with the blockchains. The **EthereumMonitor** and **BitcoinMonitor** send the current block numbers to a message queue as soon as they see them on the chains. Thereby, the rates application can always consume the current block number from the queue. There are two independent queues, one for each blockchain. If there is no new block number message on the queue the latest number is used.

Given the usage of messaging for providing the block number there is no immediate dependency between the monitor and the rates application. Another block number fetching program could replace the monitor in this functionality.

4.4.2 Serving rates

The rates application provides its exchange rates via a REST endpoint and over a message broker. Only the aggregated values are offered. While the REST endpoint only provides the latest rates, the way over the message queues allows for querying by a time stamp. Because the rates application will probably not have a rate entry exactly for a given time stamp the rate is fetched from a configurable interval around that time stamp. One can specify the time margins before and after the time stamp. The rates in that interval are fetched from the database and the closest to the time stamp is returned. If no rate entry can be found for the time interval (cf. Section 4.4.1 for why this could be the case) CryptoCompare is used as a fallback mechanism [43]. CryptoCompare allows fetching historical exchange rates for over 5000 cryptocurrencies for free. One can specify a time stamp and will receive the historical close price (end of the day price) of the desired cryptocurrency from the day on which that time stamp lies. If a rate has to be fetched via CryptoCompare because there is none available for a given time interval in the database, the fetched entry is returned but also stored in the database as a new **ExchangeEntryRate**. A new aggregated rate entry is also created, although the aggregation only includes one element per currency which are the just fetched rates.

4.4.3 Rates API

The following describes the single API endpoint provided by the rates module.

rates/current

- Returns the latest exchange rates found on the database for the supported cryptocurrencies.
- Method: GET

4.5 KYC

In this section, the KYC module is described in detail. The KYC module is responsible for handling the interaction with a generic KYC provider. First an overview over its functionality is given and then the implementation is described. In the end follows the documentation of the API and the database schema used by this module.

4.5.1 KYC Overview

The purpose and the idea behind the KYC application is to enable and facilitate the interaction between IConator and an external KYC provider. The module is built so that it can be used with the KYC provider of choice, provided web-based interaction is possible. Currently however, only the communication with the provider *IdNow* is implemented.

The KYC module is relatively isolated from the rest of IConator. It writes to its own table in the database and only needs read access to the investor table. Interaction with the KYC module happens over a simple API that is described in Section 4.5.5. Most of the endpoints in the API may make changes to the data in the database. As such, it is recommended to require authentication for these endpoints. An overview of the KYC module can be seen in Figure 4.3.

4.5.2 KYC Controller

The main part of the KYC module is the REST controller called **KycController** that exposes the simple REST API described in Section 4.5.5. It is the only part of the module with which one can interact directly. The **KycController** acts as a generic layer between IConator and an external KYC partner. The KYC process needs to be started manually by calling the corresponding endpoint. The KYC link can be added manually when starting the KYC or will be created automatically if it is not supplied while starting the KYC process. The controller also supports fetching completed KYC identifications from the

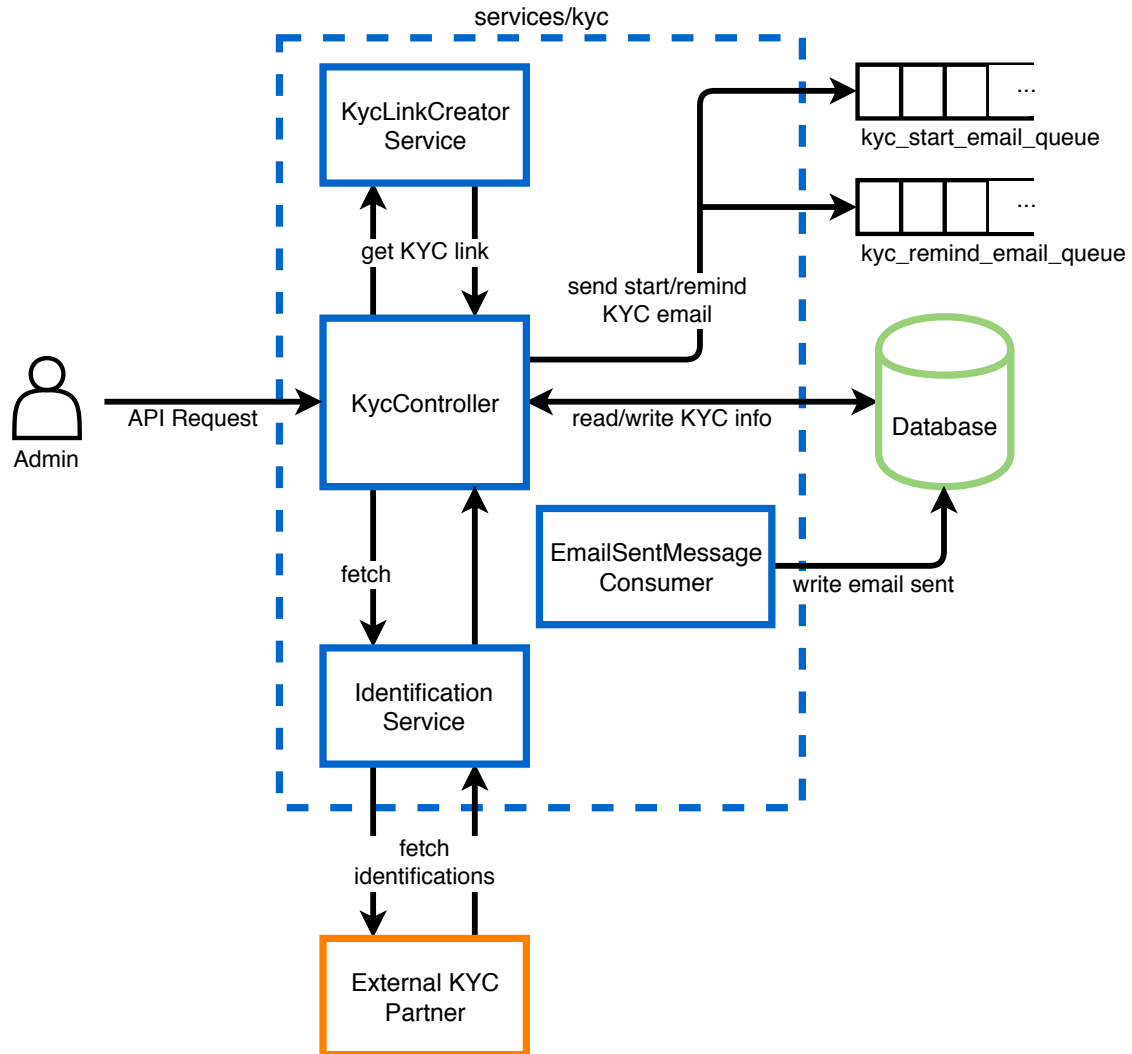


Figure 4.3: Diagram of the KYC module.

external KYC partner. Those automatically fetched identifications will be checked for successful KYC completion and the corresponding investors' KYC status will automatically be set to 'complete'.

The two automatic processes each depend on a service that needs to be implemented differently depending on the external KYC partner. Those services are described in detail in Section 4.5.3.

4.5.3 KYC Services

The KYC module uses two services for its two automatic processes. The services exist as a generic interface, so that every KYC partner may be supported by implementing the two interfaces. So far, only support for *IdNow* is implemented. The two services are described in the following.

KycLinkCreatorService

The KYC link creator service gets used when the KYC link is created automatically, i.e. when initiating the KYC, no link was provided. The service creates the KYC link for one investor. The interface provides the method `getKycLink()` that returns the created link as a string.

IdentificationService

The identification service calls the KYC provider in order to fetch all completed identifications from the provider. It only provides one method `fetchIdentifications()` that returns a list of the fetched identifications.

4.5.4 KYC Data

While the KYC module needs access to the investor table in the database in order to fetch information about the investors, it only writes to its own database table called `kyc_info`. The schema of `kyc_info` is described in Table 4.1. The primary key of the table is a UUID called `kyc_uuid` which gets created automatically by Hibernate when a new `kyc_info` entry is committed to the database. The `investor_id` is a foreign key and needs to correspond to the specific investor saved in the investor table. The entry `is_start_kyc_email_sent` is always false for a new entry and gets set to true by the message consumer `KycStartEmailMessageSentMessageConsumer` when a message in the queue is detected. The entry `no_of_reminders_sent` starts at 0 and gets incremented by the `KycReminderEmailSentMessageConsumer` when a message in its observed queue gets detected. The entry `is_kyc_complete` gets set to true when the associated identification is fetched from the KYC provider. The entry `kyc_uri` contains the KYC link that was sent to the investor in the `StartKycEmail`.

<code>kyc_uuid</code>	<code>investor_id</code>	<code>is_start_kyc_email_sent</code>	<code>no_of_reminders_sent</code>	<code>is_kyc_complete</code>	<code>kyc_uri</code>
UUID	Long	Boolean	Integer	Boolean	String

Table 4.1: Database schema of `kyc_info`.

4.5.5 KYC API

The following contains a listing of all the API endpoints of the KYC module.

`kyc/start`

- Starts the KYC process for all investors, provided they have invested and the KYC process has not already been started. The KYC URL will be generated automatically.
- Method: POST

kyc/<investorId>/start

- Starts the KYC process with the provided investor. The KYC URL will be generated automatically if not provided.
- Method: POST
- Data params:
 - Optional: Externally generated KYC URL
 - * e.g. 'http://kycprovider.com/user/123456789'

kyc/<investorId>/remind

- Sends a reminder email with the KYC link to the provided investor.
- Method: POST

kyc/fetchall

- Fetches all completed KYC identifications from the KYC provider and sets their KYC status to complete.
- Method: POST

kyc/<investorId>/complete

- Sets the KYC status of the provided investor to complete.
- Method: POST

kyc/<investorId>/status

- Returns the current KYC status of the provided investor.
- Method: GET

Chapter 5

Software Build and Deployment

An important factor to any Software Project's success is the creation of a quick and lean Software Development Life Cycle (SDLC) that enables an agile and incremental development process. In this chapter we will analyze in detail how we build, deploy, test and operate the IConator engine in different environments.

5.1 Building

ICOnator back-end [41] can be constructed and deployed both locally or remotely in a plethora of ways. IConator is built upon Java 8 [44] and Spring Boot [14] using Gradle [45].

5.1.1 Local Development

To start developing locally, clone the `ICOnator-backend` [41] repository from GitHub:

```
git clone https://github.com/ICOnator/ICOnator-backend.git
```

Then use the IDE of your preference and import the project as a Gradle project.

The module `local-dev` provides the `ICOnatorLocalDevApplication` class with a `main()` method that is able to run all the IConator applications at once, without any dependencies. It sets up an in-memory H2 database [46] and runs an Apache Qpid [47] message broker to provide for the messaging infrastructure. For usage by the email module a mock/dummy SMTP server is set up, meaning that emails are only printed to the console instead of actually being sent. An Ethereum test blockchain is also started up and connected to by the monitor module.

This functionality is provided by the Testonator project [48] which has been implemented to support the development of the IConator engine. We refer to the Testonator repository's README for instructions on how to do transactions on a running Testonator instance. In case of Bitcoin the monitor is by default configured to simply connect to real

peers on the Bitcoin network. Therefore, this will not work without a functioning internet connection.

All of the above behavior is configurable in the corresponding modules. For example the database settings would have to be changed in each module that depends on the database. Or the Ethereum blockchain that the monitor should connect to can be changed in the monitors properties file.

5.1.2 Gradle

Building with Gradle can be done simply with only one command. Navigate to the root folder of the IConator back-end project and execute the following command:

- In MacOS or Linux:
`sh gradlew clean build`
- In Windows, with CMD:
`gradlew.bat clean build`

If the build is successful, the resulting JAR files can be found under:

`services/<APPLICATION_NAME>/build/libs/`

5.2 Containerization

With the use of containers, the IConator back-end engine and the necessary components can be easily deployed in almost any environment with minimal customization.

5.2.1 Docker

All the applications are configured to be packaged as Docker [49] images. Then, it's possible to build the applications and run them independently.

Build Docker Images

You can build the images as simple as running a command in gradle. However, before building a docker image for each application (`services/*`) it's required to build the executable JARs.

In a single command line:

```
$ sh gradlew clean build && sh gradlew docker
```

By default, only the tag `latest` will be added to the docker image. If a version is required, then the project parameter `dockerVersion` should be added to gradle:

```
$ sh gradlew clean build && sh gradlew -PdockerVersion=1.0 docker dockerTag
```

This command will produce something similar to the following result:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
iconator/core	1.0	424c921ac2e9	Less than a second ago	179MB
iconator/core	latest	424c921ac2e9	Less than a second ago	179MB
iconator/email	1.0	6d489a3eda44	Less than a second ago	184MB
iconator/email	latest	6d489a3eda44	Less than a second ago	184MB
iconator/rates	1.0	87d7240a7b5a	Less than a second ago	149MB
iconator/rates	latest	87d7240a7b5a	Less than a second ago	149MB
iconator/monitor	1.0	f58d5eaf674a	Less than a second ago	180MB
iconator/monitor	latest	f58d5eaf674a	Less than a second ago	180MB

Run Docker Images

If you want to run manually, you would need to specify the environment variables provided by each application. You can get an overview of the supported environment variables by checking the ICONator- docker-compose repository.

The general command to run an ICONator application is:

```
$ SPRING_PROFILES_ACTIVE=dev docker run iconator/<APPLICATION_NAME>:latest
```

where `<APPLICATION_NAME>` is the module under `services/*`, and `SPRING_PROFILES_ACTIVE` env variable specifies the profile – which can be `"dev"` or `"prod"`. If no `SPRING_PROFILES_ACTIVE` is specified, then the default config is used.

For example, to run the `core` application:

```
$ SPRING_PROFILES_ACTIVE=dev docker run iconator/core:latest
```

Running Application's Dependencies

Message Broker: RabbitMQ

All the applications interact with AMQP protocol and require a message broker running.

The easiest way to bootstrap a message broker is to use a RabbitMQ docker image [50]. Execute the following:

```
docker run --rm -d --hostname test-rabbitmq --name rabbitmq-for-tests -e
RABBITMQ_DEFAULT_VHOST=vhost1 -p 127.0.0.1:5672:5672 -p 127.0.0.1:5671:5671
-p 127.0.0.1:15671:15671 -p 127.0.0.1:15672:15672 rabbitmq:3-management
```

SMTP server

The `email` application sends emails using SMTP and, thus, requires a server running.

The easiest way to run a private SMTP server is to setup a Google GMail Relay server [51] using plain username/password.

Once you have a GMail test account, execute the following:

```
docker run --rm -d --name smtp-server -e GMAIL_USER='<USER>'
-e GMAIL_PASSWORD='<PASSWORD>' -p 127.0.0.1:2525:25 namshi/smtp
```

where the `<USER>` and `<PASSWORD>` are, respectively, the username and password of your GMail test account.

Parity node

Run parity in the Kovan (testnet) [52] chain:

```
docker run --rm -ti -p 127.0.0.1:8180:8180 -p 127.0.0.1:8545:8545
-p 127.0.0.1:8546:8546 -p 127.0.0.1:30303:30303 -p 127.0.0.1:30303:30303/udp
parity/parity --ui-interface all --jsonrpc-interface all --tracing on
--pruning fast --warp --mode active --chain kovan
```

PostgreSQL

Run PostgreSQL with the following command:

```
docker run --rm -ti -e POSTGRES_USER=testuser -e POSTGRES_PASSWORD=testpass
-p 127.0.0.1:5432:5432 postgres
```

5.2.2 Docker Compose

The IConator Docker Compose repository provides an easy way to bootstrap the IConator services. It serves as the example to better understand the project, learn about its dependencies, and on how to deploy it.

Important: `docker-compose` version **1.13.0** or higher is required.

Run `docker-compose -version` to check if your version is compatible.

Setting an `.env` file

First, you need to create an `.env` file on the root of this repository. This file is not provided since it should contain external credentials. You should get your own credentials to run the demo.

Add the following to your `.env` file:

```

MONITOR_ETHEREUM_NODE_URL=<YOUR_ETHEREUM_NODE_URL>
GMAIL_USER=<YOUR_GMAIL_USERNAME_TO_SEND_EMAILS_FROM>
GMAIL_PASSWORD=<YOUR_GMAIL_PASSWORD>
EMAIL_SEND_FROM_ADDRESS=<YOUR_EMAIL_ADDRESS>
CORE_RECAPTCHA_ENABLED=<SUPPORT_TO_RECAPTCHA_AUTH>
CORE_RECAPTCHA_SECRET_KEY=<YOUR_RECAPTCHA_SECRET_KEY>

```

where:

- **<YOUR_ETHEREUM_NODE_URL>**: the full node URL with enabled JSON-RPC API.
- **<YOUR_GMAIL_USERNAME_TO_SEND_EMAILS_FROM>**, **<YOUR_GMAIL_PASSWORD>**: username and password of a GMail account.
- **<EMAIL_SEND_FROM_ADDRESS>**: the email address to send emails from – usually the same as **<YOUR_GMAIL_USERNAME_TO_SEND_EMAILS_FROM>**.
- **<SUPPORT_TO_RECAPTCHA_AUTH>**: a boolean that specifies whether the registration of a new investor requires recaptcha authentication.
- **<YOUR_RECAPTCHA_SECRET_KEY>**: the secret key from the ReCaptcha.

Note: Google GMail is not strictly required to run IConator. You can directly connect to any SMTP server (e.g., Amazon SES). We used Google GMail because it's easier for a demo showcase, and everyone can obtain an account.

The `docker-compose.yml`

The `docker-compose.yml` has the description of all the IConator application, the services (e.g., database, smtp server), and dependencies between them.

Also, for each application and service there are environment variables set. Change it according to your own needs.

Run

Simply run everything with:

```
$ docker-compose up -d
```

Check the logs with:

```
$ docker-compose logs -f
```

If you want to check the logs for one specific application or service (specified on the `docker-compose.yml`), run:

```
$ docker logs <SERVICE_NAME>
```

where **<SERVICE_NAME>** can be, e.g., `iconator-core` or `smtp-service`.

5.3 Cloud Deployment

Through the usage of Docker containers IConator can be deployed easily to different types of Cloud environments such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS).

For ScienceMatters' ICO (EUREKA token) we used Heroku which is PaaS, mainly due to the ease of deployment and operations in comparison to other IaaS cloud providers. This makes sense for a project that needs to be deployed fast and be operated for only 1-2 months. If IConator's back-end engine will be used for multiple ICOs either in a sequential or parallel manner, an IaaS provider might be a better option.

5.3.1 Heroku

For the IConator deployment in Heroku we tried three different methods:

- Deploying as a **Gradle App**
- Deploying as **Executable JAR files**
- Deploying as **Docker Images**

Out of these methods the easiest and most efficient one (in terms of performance) is via Docker Containers. Thus, for the Heroku platform we suggest this process for potential IConator users and describe below the detailed deployment steps as they were performed for the EUREKA token IConator back-end.

Getting Started

- Install **Heroku CLI** [53] and **Docker** locally in the operating system of your preference.
- Login to your Heroku account:
`heroku login`
- Install container plugin for your **heroku CLI**:
`heroku plugins:install heroku-container-registry`
- Login to the Heroku Container Registry [8]:
`heroku container:login`
- Create a Heroku APP for each of the IConator back-end services:
 - `heroku create <CORE-APP>`
 - `heroku create <EMAIL-APP>`
 - `heroku create <RATES-APP>`
 - `heroku create <MONITOR-APP>`
 - `heroku create <KYC-APP>`

Where `<*-APP>` is the unique heroku application identifier of your choice.

Setting Up the Services

Core Service

- Pull latest Docker image locally:
`docker pull iconator/core-master`
- Tag your docker image to your heroku app:
`docker tag iconator/core-master registry.heroku.com/<CORE-APP>/web`
- Push the docker image to heroku:
`docker push registry.heroku.com/<CORE-APP>/web`
Note: docker will only push the updated layer and common layers between services will be re-used.
- Release the docker image you pushed:
`heroku container:release web -a <CORE-APP>`
Note: You can push multiple versions of a service without actually deploying. Only after you release the image Heroku will restart your app deploying the latest image.
- Create a new postgresql database:
`heroku addons:create heroku-postgresql:<PLAN_NAME>`
Where <PLAN_NAME> is the plan of your choice.
- Set the config vars through the command-line:
`heroku config:set VAR_NAME=VALUE -a <CORE-APP>`
Note: you can also use the Heroku Web UI to add, edit or remove config vars.
- For the complete list of the config vars, their description and example values of the Core Service you can refer to Section E.1.

Email Service

- Pull latest Docker image locally:
`docker pull iconator/email-master`
- Tag your docker image to your Heroku app:
`docker tag iconator/core-master registry.heroku.com/<EMAIL-APP>/web`
- Push the docker image to heroku:
`docker push registry.heroku.com/<EMAIL-APP>/web`
Note: docker will only push the updated layer and common layers between services will be re-used.
- Release the docker image you pushed:
`heroku container:release web -a <EMAIL-APP>`
Note: You can push multiple versions of a service without actually deploying. Only after you release the image Heroku will restart your app deploying the latest image.
- Attach the postgresql database created for the Core app:
`heroku addons:attach <postgresql-APP> -a <EMAIL-APP>`
Where <postgresql-APP>, is the unique database identifier.

- Create a MailGun addon:
`heroku addons:create mailgun:starter -a <EMAIL-APP>`
- Set the config vars through the command-line:
`heroku config:set VAR_NAME=VALUE -a <EMAIL-APP>`
Note: you can also use the Heroku Web UI to add, edit or remove config vars.
- For the complete list of the config vars, their description and example values of the Email Service you can refer to Section E.2.

Rates Service

- Pull latest Docker image locally:
`docker pull iconator/rates-master`
- Tag your docker image to your Heroku app:
`docker tag iconator/rates-master registry.heroku.com/<RATES-APP>/web`
- Push the docker image to heroku:
`docker push registry.heroku.com/<RATES-APP>/web`
Note: docker will only push the updated layer and common layers between services will be re-used.
- Release the docker image you pushed:
`heroku container:release web -a <RATES-APP>`
Note: You can push multiple versions of a service without actually deploying. Only after you release the image heroku will restart your app deploying the latest image.
- Attach the postgresql database created for the Core app:
`heroku addons:attach <postgresql-APP> -a <RATES-APP>`
Where <postgresql-APP>, is the unique database identifier.
- Set the config vars through the command-line:
`heroku config:set VAR_NAME=VALUE -a <RATES-APP>`
Note: you can also use the Heroku Web UI to add, edit or remove config vars.
- For the complete list of the config vars, their description and example values of the Rates Service you can refer to Section E.3.

Monitor Service

- Pull latest Docker image locally:
`docker pull iconator/monitor-master`
- Tag your docker image to your heroku app:
`docker tag iconator/monitor-master registry.heroku.com/<MONITOR-APP>/web`
- Push the docker image to heroku:
`docker push registry.heroku.com/<MONITOR-APP>/web`
Note: docker will only push the updated layer and common layers between services will be re-used.

- Release the docker image you pushed:
`heroku container:release web -a <MONITOR-APP>`
Note: You can push multiple versions of a service without actually deploying. Only after you release the image heroku will restart your app deploying the latest image.
- Attach the postgresql database created for the Core app:
`heroku addons:attach <postgresql-APP> -a <MONITOR-APP>`
 Where `<postgresql-APP>`, is the unique database identifier.
- Set the config vars through the command-line:
`heroku config:set VAR_NAME=VALUE -a <MONITOR-APP>`
Note: you can also use the Heroku Web UI to add, edit or remove config vars.
- For the complete list of the config vars, their description and example values of the Monitor Service you can refer to Section E.4.
- Finally, setup the sale tiers even if only one sale tier is required by the ICO. Instructions for this can be found in Section 4.3.2.

KYC Service

- Pull latest Docker image locally:
`docker pull iconator/kyc-master`
- Tag your docker image to your heroku app:
`docker tag iconator/kyc-master registry.heroku.com/<KYC-APP>/web`
- Push the docker image to heroku:
`docker push registry.heroku.com/<KYC-APP>/web`
Note: docker will only push the updated layer and common layers between services will be re-used.
- Release the docker image you pushed:
`heroku container:release web -a <KYC-APP>`
Note: You can push multiple versions of a service without actually deploying. Only after you release the image heroku will restart your app deploying the latest image.
- Attach the postgresql database created for the Core app:
`heroku addons:attach <postgresql-APP> -a <KYC-APP>`
 Where `<postgresql-APP>`, is the unique database identifier.
- Set the config vars through the command-line:
`heroku config:set VAR_NAME=VALUE -a <KYC-APP>`
Note: you can also use the Heroku Web UI to add, edit or remove config vars.
- For the complete list of the config vars, their description and example values of the KYC Service you can refer to Section E.5.

Configuring IConator Custom Domains

For the services that provide an API to the front-end you need to configure a Custom Domain in your DNS provider and in Heroku.

The apps that require a subdomain to connect to the front-end are:

- **Core:**
`heroku domains:add core.yourdomain.com -a <CORE-APP>`
- **Rates:**
`heroku domains:add rates.yourdomain.com -a <RATES-APP>`
- **KYC:**
`heroku domains:add kyc.yourdomain.com -a <KYC-APP>`

Additionally, you need to Configure your application's DNS provider to point to the Heroku-supplied DNS target [54]. Lastly, confirm that your application is accessible via the custom domain. There might be a delay while DNS changes propagate.

Starting IConator on Heroku

After successfully setting up all the IConator back-end services in Heroku, configuring all the necessary add-ons and peripheral components you can start the services.

To start the respective app run:

```
heroku ps:scale web=1:standard-2x -a <*-APP>
```

Note: Starting and stopping the apps can be also performed from the Heroku Web UI. Since cloud deployment is cost-intensive it is **strongly** recommended to start/stop the services using Web UI to avoid costly mistakes and have visual feedback on the completion of your actions.

Once the app has started successfully you should see the following message:

State changed from starting to up

To stop an app run:

```
heroku ps:scale web=0 -a <*-APP>
```

Creating the Database Schema and Tables

The following action has to be performed only once after the successful IConator installation.

Important: If repeated, the Database and its contents will be reset.

Only in the **Core** app do the following:

- Set the following `config vars` to these values:
 - `JPA_GENERATE_DDL_ENABLED=true`
 - `JPA_HIBERNATE_DDL_AUTO=create`
 - `SPRING_PROFILES_ACTIVE=dev`

Every time you change a `config var` in Heroku the app restarts with the new values. Once, all three variables have been set and a successful restart occurs the database schema and tables will have been created.

- Restore the initial `config vars` values:
 - `JPA_GENERATE_DDL_ENABLED=false`
 - `JPA_HIBERNATE_DDL_AUTO=none`
 - `SPRING_PROFILES_ACTIVE=prod`

Heroku IConator Scripts

The following **bash** scripts (complete source code in Appendix D) automate some repetitive tasks performed for Heroku administration. The scripts can be executed in any Linux-based operated system that has the Bash shell environment. It is assumed that you have already installed and logged-in to *Heroku CLI*, *Heroku Container Registry* and *Docker*.

ICOnator Heroku Deploy

The script `heroku-iconator-deploy.sh` (complete source code in Appendix D.1) can be used to automatically deploy the latest IConator images to the corresponding Heroku apps.

If you run the script with no option specified, the newest images will be pushed to Heroku but will not be deployed:

```
$ ./heroku-iconator-deploy.sh
```

To release the images you can specify the `-r` option:

```
$ ./heroku-iconator-deploy.sh -r
```

ICOnator Configuration Backup

The script `heroku-config-backup.sh` (complete source code in Appendix D.2) auto-detects `*.env` files in located in the same directory. For each of these files the naming should be the unique Heroku app name, e.g., if your app is `unique-app-385` then you environment file should be `unique-app-385.env`.

For every file the script will create a `.log` file with the same naming (e.g. `unique-app-385.log`) and store all the config variables and their values in the format of `CONFIG_VAR=value` with a timestamp of the backup. In case the backup files already exist, the script will append the new backup to the end of the file.

Perform a backup:

```
$ ./heroku-config-backup.sh
```

ICOnator Configuration Update

The script `heroku-config-update.sh` (complete source code in Appendix D.3) auto-detects `*.env` files in located in the same directory. For each of these files the naming should be the unique Heroku app name, e.g., if your app is `unique-app-385` then you environment file should be `unique-app-385.env`. For every file the script will update the Heroku config variables that are specified in the file.

All the config variables and their values should be stored in the format of `CONFIG_VAR=value` with each variable/value pair in a new line. The script will only update (or create if they do not exist) the config variables that are specified in the file, it will not modify or delete any other variables.

Perform an update:

```
$ ./heroku-config-update.sh
```

Improving JVM Performance

Since the operation of a software engine in the cloud can add up significantly in costs, you can optimize the JVM options [55] to reduce the need for resources and drop the overall cost.

In all 5 IConator apps you have to change the `JAVA_TOOL_OPTIONS` variable (Heroku specific) to the following value: `-Xms256m -Xmx800m -Xss1m`

- `Xms256m` sets the initial memory allocation pool of the JVM to 256 megabytes.
- `Xmx800m` sets the maximum memory allocation pool of the JVM to 800 megabytes.
- `Xss1m` sets the size of the frame stack used by each thread to 1 megabyte.

Of course, these are indicative values which have resulted from our own tweaking tests using Heroku. You can use the Application Performance Monitoring (APM) solution of your choice to monitor each app's performance and configure the JVM options accordingly based on your needs.

Useful Heroku Commands

- List all config variables set by the user:
`heroku config -a <*-APP>`
- List all addons for an app:
`heroku addons -a <*-APP>`
- List all variables for an app including config and env variables:
`heroku run printenv -a <*-APP>`

5.4 Operations

5.4.1 Key Generation

The key generation scripts and processes described below have been developed by the IConator-cmd project [56], primarily developed and maintained by Prof. Dr. Thomas Bocek and Dr. Guilherme Sperb Machado.

An important process for an ICO with IConator is to generate the Bitcoin and Ethereum public and private keys for investors to pay in. It is extremely important to keep the private keys secure, as they will be worth potentially millions. From previous ICOs we came up with the following best practice process. This process requires at least 3 people (Alice, Bob, Carol). Carol can be one or more persons.

Preparations

- For the key generation, Alice needs to have git and go installed on their machine.
- As well as one Laptop, new 4 USB sticks from different brands, and a hardware wallet.
- For the consolidation, a Bitcoin core full client and a Geth full client [57] are necessary. Syncing may take a few days, so this needs to be done beforehand.
- **Important:** sanity test and getting familiar with the tools. Download and build all command line tools. Generate 5 keys and paying small amounts of Bitcoin and Ethers in the mainnet. Run the consolidation script that will transfer these Bitcoins and Ethers back to your account. Make sure the coins arrive and can be further used (e.g., transfer it to another account). Once you have verified that the funds are all there and transferable (minus fees), you are familiar with the process.

Generate private and public keys

1. Buy a laptop that was never connected to the Internet, don't connect it to the Internet! Alice buys the laptop, while Bob unseals it. Unsealing is supervised by Alice (and Carol).
2. Setup the laptop. Setup is done by Alice, supervised by Bob (and Carol). A password for the laptop is only known by Alice and Bob. If a bootable stick with Linux is used, creating the stick is done by Alice, while Bob (and Carol) checks if the image on the stick matches the Linux image. For the comparison, `cmp /tmp/ubuntu.img /dev/sdX -n 500GB` can be used.
3. Alice checks out this repository and builds keygen with. This process is supervised by Bob (and Carol):

```
git clone https://github.com/ICOnator/ICOnator-cmd.git
cd ICOnator-cmd/key800
go get && go build
```
4. If the binaries support reproducible builds, the checksum of the binaries are compared. If not, the source code has to be checked by Bob (and Carol), and Bob builds the binaries supervised by Alice (and Carol).
5. Alice copies the binaries to a USB stick and copies it to the new Laptop.
6. Bob verifies the checksum of the binary on the new laptop matches with the newly build binaries. This process is supervised by Alice (and Carol).
7. Alice runs the key generation script with 100'000 keys and specifies a file to store the private and public keys.
8. Bob copies the private keys to the 2 USB sticks, and public to the remaining stick. The public key stick has to be visibly marked as the one with the public keys to avoid mixing them up.
9. Alice, Bob (and Carol) go to the bank vault and safely store the laptop and the 3 USB keys with the private keys in the bank vault. To access the bank vault, Alice and Bob are required to go there.
10. The stick with the public key can now be used for the paying in ICOnator.

5.4.2 Key Consolidation

1. Make sure your laptop has the Bitcoin core full client and Ethereum full client installed. Since this laptop is connected to the Internet, make sure it's not compromised. As no private keys are stored withing the Ethereum full client, the Ethereum full client can be installed on one machine, while the consolidation script runs on another machine. However, private keys are stored in the Bitcoin core client, thus it is important to be sure that the machine is not compromised.
2. Alice checks out this repository and builds consolidate-eth and consolidate-btc with. This process is supervised by Bob (and Carol):

```
git clone https://github.com/ICOnator/ICOnator-cmd.git
cd ICOnator-cmd/consolidate-eth
go get && go build
cd ../consolidate-btc
go get && go build
```

3. If the binaries support reproducible builds, the checksum of the binaries are compared. If not, the source code has to be checked by Bob (and Carol), and Bob builds the binaries supervised by Alice (and Carol).
4. Alice, Bob (and Carol) go to the bank and get one USB stick, the rest stays in the vault.
5. The private keys are copied to the laptop with the consolidation binaries by Bob. The scripts are run sequentially by Alice. Alice and Bob (and Carol) will supervise the process. The private keys, hardware wallet, and laptop are always in a room with at least two people.
6. The consolidation address is generated from the hardware wallet. Setup first the hardware wallet (see Hardware Wallet Setup).
7. Alice copies addresses from Trezor and save somewhere (needed for contract owner, consolidation addresses). Bob (and Carol) validate address's checksum by validating it with e.g., MyetherWallet. Unplug the hardware wallet.
8. Bob runs the consolidation scripts with the consolidation address from the hardware wallet. This will take some time. After consolidation Alice, Bob (and Carol) check if all funds have been transferred, watch out for error messages.
9. Alice laminate seed on paper and PIN and bring to bank vault, bring the hardware wallet and the private keys back to the vault as well. There may be Airdrops or coins due to forks on these accounts. To access the funds, Bob and Alice need to go the bank vault, get the Trezor and bring it back again.

Hardware Wallet Setup

1. Bob: Reset Trezor (Supervised by Alice and Carol)
2. Bob: Initialize Trezor (Supervised by Alice and Carol)
3. Bob: Set PIN (Supervised by Alice, no one else present)

4. Alice: Unlock Trezor with PIN
5. Carol: Write down Seed (mnemonic 24-word seed)
6. Carol and Bob: Cross check seed
7. Carol: Laminate sheet
8. Bob: Send Public Addresses from Trezor to Alice (supervised by Carol and Alice) needed for contract owner and consolidation

Chapter 6

Software Evaluation and System Analysis

This chapter evaluates the quality of the ICONator’s source code in Section 6.1 and provides a robustness analysis of the ICONator as a running system in Section 6.2. A more meticulous evaluation of the ICONator with a systematic test setup is not part of this project. So far, the ICONator has been successfully tested for its functional requirements. The engine has been deployed several times and investment scenarios with Bitcoin and Ether have been tested successfully on the respective test networks. In future work a methodical, practical evaluation, especially of performance and security, has to be done to prove the ICONator’s resilience to attacks and high workloads.

The ICO of the EUREKA token project [27] was foreseen to be the first opportunity to prove that the ICONator works in production. The engine was ready and an investor web front-end was already in place. But in the end the ICO didn’t take place due to reasons beyond our control. Although, it is known that the ICONator was used in the ICO of the DEZOS project [58]. No disruptions or errors were reported to us in the course of this project. Due to this and our manual tests, we claim that the ICONator is production-ready.

6.1 Software Quality

For the development of the ICONator, the latest technologies and tools, such as Spring Boot, Docker(-Compose), Git(Hub), TravisCI, Heroku and the ELK stack were used, to provide a plethora of options to build, compile and run the ICONator engine in virtually any environment with minimal migration effort. The usage of widespread open-source frameworks and deployment tools should allow new developers to continue the development with a flat learning curve.

In order to oversee the quality of the software contributed by multiple independent developers, a tool that monitors the code and maintains some quality standard is needed. Software quality focuses on detecting violations of rules for good architectural and good coding practices [59]. In this project the static code analysis tool SonarQube [60] was used for this purpose. In static code analysis the source code and project structure is analyzed directly, and not as a running program. In case of SonarQube it can detect security vulnerabilities and programming

errors that might lead to exceptions at runtime, it makes recommendations on code formatting according to popular conventions, and it computes common metrics, such as cyclic complexity, that help directing the efforts of the developer. Developers that carry on the development for the IConator should stick to the usage of SonarQube to support adherence to coding conventions and allow for continuous tracking of the code quality.

Section 6.1.1 briefly discusses the IConator's code coverage and Section 6.1.2 mentions considerations about SonarQube's findings from the security perspective.

6.1.1 Code Coverage

The code coverage measures how much of the productive code is covered by unit tests. High code coverage means that unit tests have passed through most of the lines of code. Though, this doesn't directly imply that all intended functionality is proven. Unit tests have to be carefully thought through in order to guarantee that the written code is doing what it should and can handle corner cases correctly.

The IConator's code coverage, evaluated with SonarQube, was at approximately 40% at the time of writing. This shows that future work should include the implementation of more unit tests covering the IConator's functionality. There are code parts with high cyclomatic complexity (up to 55) that demand a lot more test cases than others to cover all the possible paths through the code. A detailed view on which code locations need the most attention in regard to code coverage can be obtained by fetching the IConator's source code and running SonarQube on it.

6.1.2 Security

SonarQube reports vulnerabilities and categorizes them into either major or minor depending on their conformity to the top security vulnerabilities list from two different institutions. One is the top 10 list of the Open Web Application Security Project (OWASP) [61], the second one is the top 25 list of the SysAdmin, Networking and Security (SANS) institute [62].

All the IConator's endpoints, the code locations in which files are read, and entry points at which command line arguments are processed are marked as security hotspots by SonarQube. The endpoint security is handled by Spring Boot. It provides configuration mechanisms for restricting access to certain endpoints. An IConator operator can have different needs for the endpoint security configuration and will need to apply changes directly in the source code. Sanitization of input coming through the endpoints is already done through the definition of the expected request formats. Again, the Spring framework does most of the work. Regarding program arguments, no special measures have been taken to secure the affected code parts. The only users that will have an influence on those arguments are the operators of the engine.

All locations pointed out by SonarQube that correspond to file handling are not vulnerable to malicious user inputs, e.g. such that could lead to resource exhaustion. Therefore, they are not specially secured.

6.2 Robustness

Because the IConator handles financial data, robustness plays an important role. Although this requirement mainly applies to the monitor application, this section evaluates the robustness of all five applications. Here, the definition of robustness is constrained to tolerance against system failures. With system failures the sudden and prolonged loss of connectivity to one of the applications is meant. Be it because of a network failure or a failure of the system the application is running on. Sections 6.2.1 and 6.2.2 describe the impact of a failure on the affected application itself and on the other running applications. Ensuring correct handling of faulty and unexpected inputs or of failures within the code is a task for unit testing.

6.2.1 Applications

Core

If the core application crashes, all the endpoints used by investors and administrators become unavailable. For example new investors can't register and existing investors cannot check their payment information. Though, incoming payments of already registered investors will still be handled normally because the core application does not handle that part. Also, confirmation and token allocation emails will still be sent out.

A simple restart of the core application is enough for recovery from the failure. Additionally, if possible unavailability is not acceptable, two core application instances can be run concurrently with a load balancer distributing requests to the endpoints between them.

Email

If the email application crashes all requests for sending out emails will be stale as long as the application is down. But because the email application creates message queues that are not deleted when it crashes, after a restart all messages that went into the queue during down time can be consumed and the corresponding emails will be sent out.

Therefore, an unavailable email application only creates a delay in sending the email messages. If this delay is not acceptable, the email application can be run in multiple instances concurrently. The instances will each consume from the same queue. Each message in the queue is only processed by one of the instances, and if one instance fails the others will continue consuming message and sending emails.

Rates

A running rates application is crucial for proper processing of incoming transactions. As soon as the rates application crashes, the monitor will fail in trying to receive exchange rates. It will create refund entries for all the transactions that occur while the rates application is down. The monitor does currently not have a retry mechanism which would try to reprocess these transactions and eventually succeed as soon as the rates is up and running again.

It is actually more desirable that the monitor and the rates application stop running at the same time. In this scenario the monitor cannot create any refund entries for transactions that arrive during down time. When restarting the two applications, one must make sure that the rates application is running before the monitor picks up his work. Thereby, it is guaranteed that the monitor's rates requests will be answered by the rates application and transactions can be fully processed.

The second impact of a crashed rates application are missing exchange rate entries in the time interval in which the application was not running. This is a lesser problem because the missing values can be fetched from the CryptoCompare service if needed.

To prevent the impacts of an unavailable rates application, multiple rates application instances can be run concurrently. They can use the same database for storing the exchange rates. They will both write new exchange rate entries to the database in the configured intervals, which can be different in each instance. The rate requests of the monitor will be answered by one of them, because they consume these requests from the same queue. In this redundant setup an unavailability of one of the instances will not impact the monitor. Because the rates application also has an endpoint which allows fetching the current exchange rates one could also add a load balancer in front of the instances to distribute the calls to that endpoint between them.

Monitor

The monitor must ensure consistency of all investments. It was designed so that it can always determine how a given transaction has to be handled. Be it a new transaction, a transaction that was only partially processed due to a application crash, or a transaction without all needed information to it, the monitor will always make sure that the investor is either allocated the appropriate amount of tokens or a corresponding refund entry is created. The next paragraph goes through different scenarios and explains how they are handled.

When the monitor application crashes, processing of transactions that are made between crash and restart is delayed. But as soon as the monitor is running again it will scan the blockchains, find the transactions, and process them correctly. If a crash interrupts the processing of a transaction, it is left in an incomplete state. After a restart the monitor scans the blockchain and picks that transaction up again. It sees its incomplete state in the database and finishes the processing as needed. No duplicate token allocations happen and investors are always informed about the outcome of their investment (if the email application is running). If a new investor registers in the time between a crash and restart of the monitor, that investor is simply detected at startup because the monitor reads the investors from the database. If such an investor has also made an investment at downtime, the transaction will be successfully detected on startup too.

In the time in which the monitor is unavailable incoming transactions are not immediately processed. If that delay is unacceptable multiple monitor instances can be run concurrently. If one becomes unavailable the other instance will continue processing transactions on time. Section 4.3.3 discusses concurrently running monitor instances in detail.

KYC

The KYC component is isolated from the rest of the ICONator components and only depends on the message queue for communication with the email module. If there was an error with

sending an email, the KYC application gets informed by that. All interaction with the KYC module is manual via its API. A crash of the KYC module has no impact on any of the other components. A simple restart of the module will suffice after a crash.

6.2.2 System Context

Message Broker

The message broker orchestrates the communication between the IConator applications. It therefore constitutes a critical part without which many functionalities are incapacitated. Some of them are mentioned in the following:

- An investor registers and the core application stores the investors' information in the database, but the monitor doesn't get notified about the new investor because that usually happens via the message broker. The monitor would have to be restarted to read the new investor information from the database.
- The monitor can't get exchange rates from the rates application. The rates application can't get block numbers from the monitor.
- Non of the applications can send emails because the email application doesn't receive any messages anymore.

As a mitigation to this problem the RabbitMQ broker can be deployed on a cluster which provides higher availability than just running on one node.

Database

The database is used by all applications but the email application. If it is unavailable errors will occur and will be logged. There is no mechanism with which the applications can recover from a failed attempt to read from or write to the database. The current processes will throw an exception and continue from where it is caught. Future work could include a retryer at critical locations to introduce more robustness to database failures.

To evade the impact of an unavailable database, the database can be made redundant in a master-slave setup with replicas being ready for fail-over, in case the master node fails.

SMTP Server

The SMTP server is only used by the email application. If it is unavailable the email application will run into an error and throw an exception. In future the email sending should be handled by a retryer to avoid losing messages due to short unavailabilities of the SMTP server. Additionally, the email application could be extended so that it informs the requesting application about failures. Currently this is only the case for the KYC module. Multiple SMTP servers behind a load balancer are also a remedy for the possible unavailability of the SMTP server.

Currency Exchanges

Currency Exchanges are consulted by the rates application for exchange rates. The IConator works with six different exchanges. An operator can configure which of these exchanges should be used for retrieving exchange rates. He should definitely configure more than one to ensure high availability. That way, rates application is robust against failure of reaching single currency exchanges.

Blockchain Nodes and Peers

The monitor application retrieves information about the Ethereum chain from an Ethereum full node. Which node to connect to is configurable. If that node crashes, the monitor will not notice new transactions until the node is restarted. The new transactions will be retrieved by monitor as soon as the node starts working again. Multiple Ethereum nodes behind a load balancer could be used as a remedy to this problem. The monitor's requests would be distributed between the nodes and if one becomes unavailable the other one will go on responding with the newest transactions.

In the case of Bitcoin, there is no single point that can stop the monitor from receiving new block information. Multiple peers are available at all times.

Chapter 7

Summary and Final Considerations

7.1 Summary

With the possibility of selling digital tokens to investors for raising initial capital, ICOs have become very popular especially among tech start-ups. At the same time, the need for platforms and services providing the necessary functionality and support for running an ICO was met by multiple providers mentioned in Section 2.2. But among all the paid products, a free, open-source and ICO-ready software package was missing. The vision of such a product was the starting point of this project. More precisely, an ICO engine that can be configured, deployed, and operated by anyone with the need to run an ICO. For this purpose, the already existing ICONator project was picked up and refined.

The existing source code was examined so that the tasks of the modules and applications could be explicitly formulated. This was followed by the design and implementation of new components, as well as, the redesign and refinement of existing parts. A new KYC component was added that introduces support for interaction with external KYC partners. This was an important step towards functionalities that help ICO operators with legal compliance. The existing rates and monitor applications were redesigned with reliability and robustness in mind. The transaction monitoring and processing were completely re-implemented and the concept of sale tiers was introduced. Unit tests were added to cover the core functionalities of the new and reworked code parts.

Apart from the code-based contributions, ICONator was envisaged to be easily deployable on different platforms, for this purpose Docker and containerization were utilized. The ICONator's applications have been packaged into separate Docker images and are available on Docker Hub. Heroku was used as an example deployment environment in which all manual tests were conducted. This demonstrated the compatibility and ease of operability of ICONator hosted on a modern cloud platform.

Furthermore, this report was created, which serves as an in-depth documentation on how the ICONator works. On one hand, it contains information on how to set up and operate the ICONator and, on the other hand, it gives future developers insights into considerations and decisions that were made regarding the design and implementation of the engine.

The ICONator's first real-world mission was envisaged to be the ICO of the EUREKA token. Unfortunately, due to reasons beyond our control, that ICO did not take place. Therefore, no

direct experience with the engine in production could be gathered in this project. But manual tests were conducted to ensure the proper functioning of the engine. Furthermore, the DEZOS project successfully conducted their ICO with the ICONator, which proves that it is production-ready.

7.2 Final Considerations

ICOs offer a way for a company to collect funding and an engine like the ICONator is a tool that supports and automates such an undertaking. However, setting up and deploying an ICO engine is not a simple task and can in itself incur high costs. There are some points that need to be considered before deciding on using the ICONator.

First, technical knowledge of the team behind the ICO needs to be considered. The ICONator makes use of many different technologies, of which each can come with their own set of challenges. Decisions have to be made on how and where to deploy the engine, what fallbacks are needed and if there is a need for multiple instances running in parallel. Each of those decisions influence the costs caused by the ICO. The right decisions will depend on the specific environment of the ICO. One possibility that could reduce the costs incurred by the deployment and the technical team behind it, is to allow multiple ICOs running on the same ICONator instance. The companies would be able to pool their resources and investors investing in multiple ICOs would only need to complete the KYC once. However, this possibility is not yet supported by ICONator and would introduce additional complexity in the architecture.

Second, the legal implications brought up by an ICO need to be considered. During an ICO, many transactions of monetary value take place and, depending on the country the ICO is taking place in, this means that a lot of rules and regulations have to be followed. Neglecting those laws could have far-reaching consequences for the issuer of the ICO. Just because it is possible to set up and deploy the ICONator, this does not mean that the planned ICO can just be started without any additional work. In many countries, adherence to anti-money laundering policies is required. The ICO may need approval from a governmental organization and a KYC process needs to be implemented. Some countries even outright ban ICOs.

Third, marketing is of utmost importance. An ICO is of no use when hardly anyone knows about it. Consider a deployed ICONator where all technical decisions are already made and all legal issues are solved. If no one knows about the ongoing ICO, it will end up being a big time and money sink. Thus, a successful ICO does greatly depend on the marketing done beforehand in order to make the company and its ideas and vision known to the world.

7.3 Future Work

Throughout this report, improvements and additional features were mentioned that should receive attention, if work on ICONator continues in the future. What follows is a complemented list of these points. Additionally, there are open issues that can be found on the ICONator's GitHub repository. These partly overlap but also complement what is listed here.

- This Master Project could not closely test the ICONator in a real-world scenario. The engine, however, is production-ready and it is known that ICONator was used in other

ICO projects (e.g., DEZOS [58]). Following along a real-world ICO that is using ICONator could offer great insights into which areas of the engine need refinement and which parts are working great.

- Application properties management is one aspect that could be significantly improved for each individual app and for the ICONator in general. Offering a centralized and easy-to-operate way to configure the engine would be desirable.
- Monitoring a running ICO, e.g. checking the state of its sale tiers or the number of registered investors, is no comfortable task at the moment. Information has to be retrieved from the database directly. A graphical interface for the operator would enhance usability a lot, although one might not include such a feature in the definition of an engine.
- The ICONator's apps are decoupled and communicate with each other through the message broker. One way to introduce authentication and non-repudiation between the different components would be to sign the messages. This would add a layer of security to the communication happening between the apps and hinder attackers to inject malicious messages unnoticed.
- Support for additional cryptocurrencies would make the engine more attractive to use by reducing the barrier of entry for users that would first need to convert their cryptocurrency to Ethereum or Bitcoin. Additionally, by supporting ERC20 tokens, holders of all ERC20 based currencies could take part in an ICO with ICONator.
- If the processing of a transaction fails because the exchange rate could not be fetched, the monitor application simply creates a refund entry in the database. A retry mechanism is more desirable for this case. The availability of the rates application may be compromised only for a short time.
- As mentioned in Section 6.1.1, improving code coverage should be part of future work.

Bibliography

- [1] Coindesk: \$6.3 Billion: 2018 ICO Funding Has Passed 2017's Total. Available at: <https://www.coindesk.com/6-3-billion-2018-ico-funding-already-outpaced-2017>. Last visited on: 23.11.2018.
- [2] IConator website. Available at: <https://iconator.io/>. Last visited on: 03.01.2019.
- [3] Modum website. Available at: <https://modum.io/>. Last visited on: 03.01.2019.
- [4] Elastic: ELK stack. Available at: <https://www.elastic.co/elk-stack>. Last visited on: 03.01.2019.
- [5] Heroku website. Available at: <https://www.heroku.com/>. Last visited on: 03.01.2019.
- [6] Heroku: Deploying Gradle Apps on Heroku. Available at: <https://devcenter.heroku.com/articles/deploying-gradle-apps-on-heroku>. Last visited on: 03.01.2019.
- [7] Heroku: Deploying Executable JAR Files. Available at: <https://devcenter.heroku.com/articles/deploying-executable-jar-files>. Last visited on: 03.01.2019.
- [8] Heroku: Container Registry & Runtime (Docker Deploys). Available at: <https://devcenter.heroku.com/articles/container-registry-and-runtime>. Last visited on: 03.01.2019.
- [9] RabbitMQ website. Available at: <https://www.rabbitmq.com/>. Last visited on: 03.01.2019.
- [10] Mailgun: Transactional Email API Service For Developers. Available at: <https://www.mailgun.com/>. Last visited on: 03.01.2019.
- [11] PostgreSQL website. Available at: <https://www.postgresql.org/>. Last visited on: 03.01.2019.
- [12] EUREKA Token Sale website. Available at: <https://buy-dev.eurekatoken.io/>. Last visited on: 03.01.2019.
- [13] Bash GNU Project Free Software Foundation website. Available at: <https://www.gnu.org/software/bash/>. Last visited on: 03.01.2019.
- [14] Spring Boot website. Available at: <http://spring.io/projects/spring-boot>. Last visited on: 03.01.2019.
- [15] Docker Compose website. Available at: <https://docs.docker.com/compose/>. Last visited on: 03.01.2019.

- [16] SAFT Project website. Available at: <https://saftproject.com/>. Last visited on: 18.01.2019.
- [17] ICO Launch Malta: ICO Platform website. Available at: <https://icomalta.com/>. Last visited on: 06.01.2019.
- [18] Eidoo website. Available at: <https://icoengine.net/>. Last visited on: 06.01.2019.
- [19] Das33 website. Available at: <https://das33.io/>. Last visited on: 06.01.2019.
- [20] Coinfactory website. Available at: <https://icoplatform.tech/>. Last visited on: 06.01.2019.
- [21] ICOBox website. Available at: <https://icobox.io/>. Last visited on: 06.01.2019.
- [22] ICO Platform website. Available at: <https://www.icoplatform.io>. Last visited on: 06.01.2019.
- [23] ICO Platform website. Available at: <https://theicoplatform.com/>. Last visited on: 06.01.2019.
- [24] Tokengate website. Available at: <https://tokengate.io/>. Last visited on: 13.02.2019.
- [25] Token Wizard website. Available at: <https://wizard.oracles.org/>. Last visited on: 06.01.2019.
- [26] TokenMarket website. Available at: <https://tokenmarket.net>. Last visited on: 06.01.2019.
- [27] EUREKA by ScienceMatters website. Available at: <https://www.eurekatoken.io/>. Last visited on: 07.01.2019.
- [28] ScienceMatters website. Available at: <https://sciencematters.io/>. Last visited on: 07.01.2019.
- [29] CloudAMQP: Johansson Lovisa - What is message queuing?. Available at: <https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html>. Last visited on: 03.12.2018.
- [30] Amazon AWS: What are the benefits of message queues?. Available at: <https://aws.amazon.com/message-queue/benefits/>. Last visited on: 27.11.2018.
- [31] AMQP: AMQP is the Internet Protocol for Business Messaging. Available at: <https://www.amqp.org/about/what>. Last visited on: 27.11.2018.
- [32] Thymeleaf website. Available at: <https://www.thymeleaf.org/>. Last visited on: 23.08.2018.
- [33] Github: Bitcoinj, a library for working with Bitcoin. Available at: <https://bitcoinj.github.io/>. Last visited on: 08.01.2019.
- [34] Web3j: Lightweight Java and Android library for integration with Ethereum clients. Available at: <https://web3j.io/>. Last visited on: 08.01.2019.
- [35] Tanzo website. Available at: <https://tanzo.io/>. Last visited on: 08.01.2019.

- [36] Medium: nOS Token Sale Details Instructions. Available at: <https://medium.com/nos-io/nos-token-sale-details-instructions-ecee9ccdae1c>. Last visited on: 08.01.2019.
- [37] VISA: Fact Sheet. Available at: <https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf>. Last visited on: 07.11.2018.
- [38] Bitcoin Wiki: Scalability. Available at: <https://en.bitcoin.it/wiki/Scalability>. Last visited on: 07.11.2018.
- [39] Ethereum StackExchange: How many transactions per second can Ethereum currently handle? What changes will allow the network to be able to handle more?. Available at: <https://ethereum.stackexchange.com/questions/49484/how-many-transactions-per-second-can-ethereum-currently-handle-what-changes-wil>. Last visited on: 07.11.2018.
- [40] Draw.io website. Available at: <https://www.draw.io/>. Last visited on: 08.01.2019.
- [41] GitHub: IConator back-end Repository. Available at: <https://github.com/ICOnator/ICOnator-backend>. Last visited on: 03.01.2019
- [42] XChange website. Available at: <https://knowm.org/open-source/xchange/>. Last visited on: 08.01.2019.
- [43] Cryptocompare API. Available at: <https://min-api.cryptocompare.com/>. Last visited on: 08.01.2019.
- [44] Java 8 website. Available at: <https://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html>. Last visited on: 03.01.2019.
- [45] Gradle website. Available at: <https://gradle.org/>. Last visited on: 03.01.2019.
- [46] H2 Database website. Available at: <http://www.h2database.com/html/main.html>. Last visited on: 03.01.2019.
- [47] Apache Qpid website. Available at: <https://qpid.apache.org/>. Last visited on: 03.01.2019.
- [48] GitHub: IConator Testonator Repository. Available at: <https://github.com/ICOnator/Testonator>. Last visited on: 03.01.2019.
- [49] Docker website. Available at: <https://www.docker.com/>. Last visited on: 03.01.2019.
- [50] DockerHub: RabbitMQ Container Repository. Available at: https://hub.docker.com/_/rabbitmq/. Last visited on: 03.01.2019.
- [51] Google: SMTP relay - Route outgoing non-Gmail messages through Google. Available at: <https://support.google.com/a/answer/2956491?hl=en>. Last visited on: 03.01.2019.
- [52] GitHub: Kovan Testnet The Fast and Reliable Ethereum Test Chain. Available at: <https://kovan-testnet.github.io/website/>. Last visited on: 03.01.2019.
- [53] Heroku: The Heroku CLI. Available at: <https://devcenter.heroku.com/articles/heroku-cli>. Last visited on: 03.01.2019.
- [54] Heroku: Custom Domain Names for Apps. Available at: <https://devcenter.heroku.com/articles/custom-domains>. Last visited on: 03.01.2019.

- [55] Heroku: Java Support | Environment. Available at: <https://devcenter.heroku.com/articles/java-support#environment>. Last visited on: 03.01.2019.
- [56] GitHub: IConator-cmd Repository. Available at: <https://github.com/IConator/IConator-cmd>. Last visited on: 03.01.2019.
- [57] Ethereum: Go Ethereum Official Go implementation of the Ethereum protocol. Available at: <https://geth.ethereum.org/>. Last visited on: 03.01.2019.
- [58] DEZOS website. Available at: <https://dezos.io/>. Last visited on: 15.02.19.
- [59] CISQ website. Available at: <https://it-cisq.org/>. Last visited on: 26.01.19.
- [60] Sonarqube website. Available at: <https://www.sonarqube.org/>. Last visited on: 26.01.19.
- [61] OWASP website. Available at: https://www.owasp.org/index.php/Main_Page. Last visited on: 26.01.19.
- [62] SANS website. Available at: <https://www.sans.org/>. Last visited on: 26.01.19.

Abbreviations

AML	Anti Money Laundering
APM	Application Performance Monitoring
AMQP	Advanced Message Queuing Protocol
DNS	Domain Name System
ELK	Elasticsearch Kibana Logstash
IaaS	Infrastructure as a Service
ICO	Initial Coin Offering
IPO	Initial Public Offering
JAR	Java Archive
JVM	Java Virtual Machine
KYC	Know Your Customer
PaaS	Platform as a Service
SaaS	Software as a Service
SAFT	Simple Agreement for Future Tokens
SDLC	Software Development Life Cycle
SMTP	Simple Mail Transfer Protocol

List of Figures

3.1	System Context of IConator. Source: figure adapted from Dr. Guilherme Sperb Machado's presentation.	12
3.2	Platform architecture with simplified message queue. Source: figure adapted from Dr. Guilherme Sperb Machado's presentation.	13
3.3	Routing keys used by each application to send messages to the exchange.	16
3.4	Message queues used by each application to consume messages.	17
4.1	Diagram of the email module.	23
4.2	Diagram of the monitor application and its surrounding systems.	26
4.3	Diagram of the KYC module.	39

List of Tables

2.1	Overview of ICO platforms.	8
3.1	Overview of all routing keys used by IConator.	15
4.1	Database schema of <code>kyc_info</code>	40

Appendix A

Contents of the CD

- A copy of all ICONator GitHub Repositories:
 - Testonator,
 - ICONator-backend,
 - ICONator-docker-compose,
 - ICONator-cmd,
 - ICONator-FAQ
- A PDF file of the Master Project report
- The LaTeX source code of the Master Project report
- Related Work websites as HTML
- The set of slides used for the final presentation

Sale tier configuration example

```
INSERT INTO sale_tier (tier_no, version, description, start_date,
    end_date, discount, tomics_max, tomics_sold, has_dynamic_duration,
    has_dynamic_max)
VALUES (1, 0, 'Crowd_1', '2018-09-01', '2018-10-01', 0.25,
    5000000000000000000000000000, 0, true, false);
```

```
INSERT INTO sale_tier (tier_no, version, description, start_date,
    end_date, discount, tomlcs_max, tomlcs_sold, has_dynamic_duration,
    has_dynamic_max)
VALUES (2, 0, 'Crowd_2', '2018-10-01', '2018-10-02', 0.20, 0, 0, true
    , true);
```

```
INSERT INTO sale_tier (tier_no, version, description, start_date,
    end_date, discount, tomics_max, tomics_sold, has_dynamic_duration,
    has_dynamic_max)
VALUES (3, 0, 'Crowd_3', '2018-10-02', '2018-10-09', 0.15, 0, 0, true
    , true);
```

```
INSERT INTO sale_tier (tier_no, version, description, start_date,
    end_date, discount, tomics_max, tomics_sold, has_dynamic_duration,
    has_dynamic_max)
VALUES (4, 0, 'Crowd_4', '2018-10-09', '2018-10-16', 0.10, 0, 0, true
    , true);
```

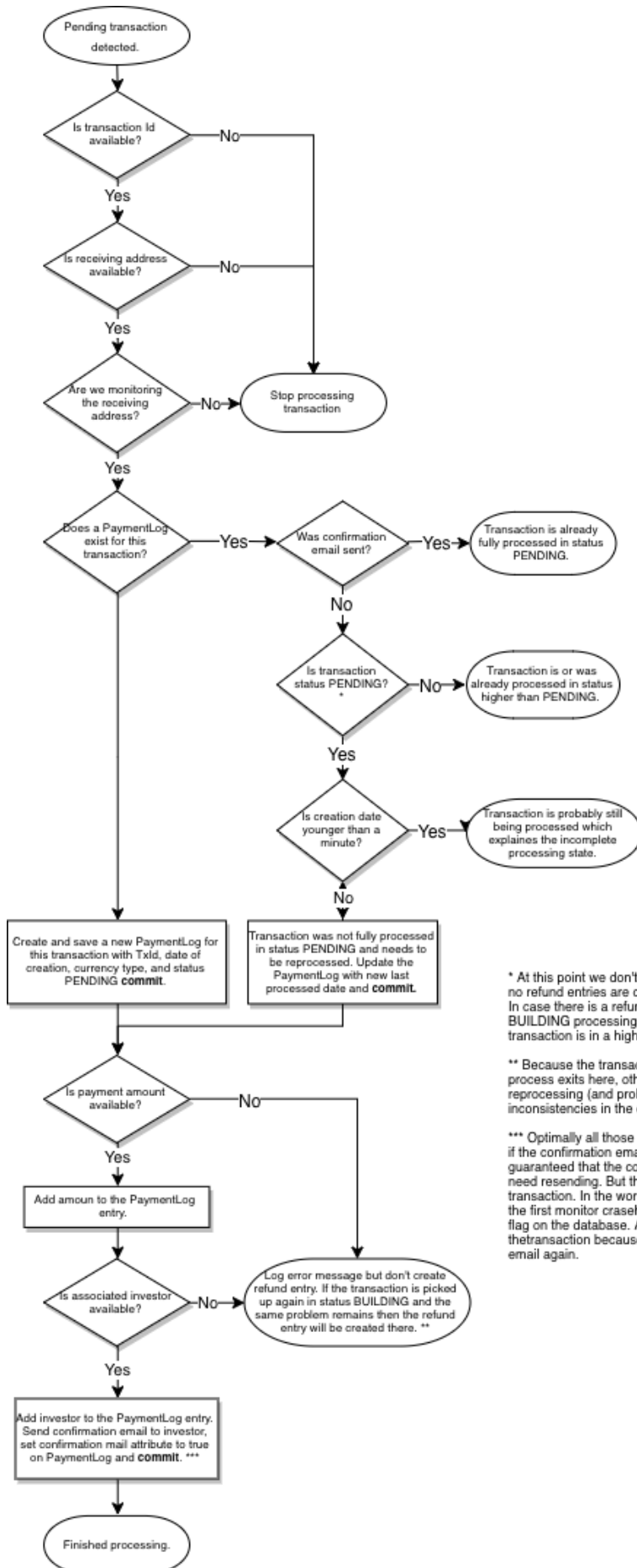
81

VALUES (5, 0, 'Crowd_5', '2018-10-16', '2018-10-23', 0.0, 0, 0, **true**,
true);

Appendix C

Transaction processing flows

C.1 Pending transactions flow



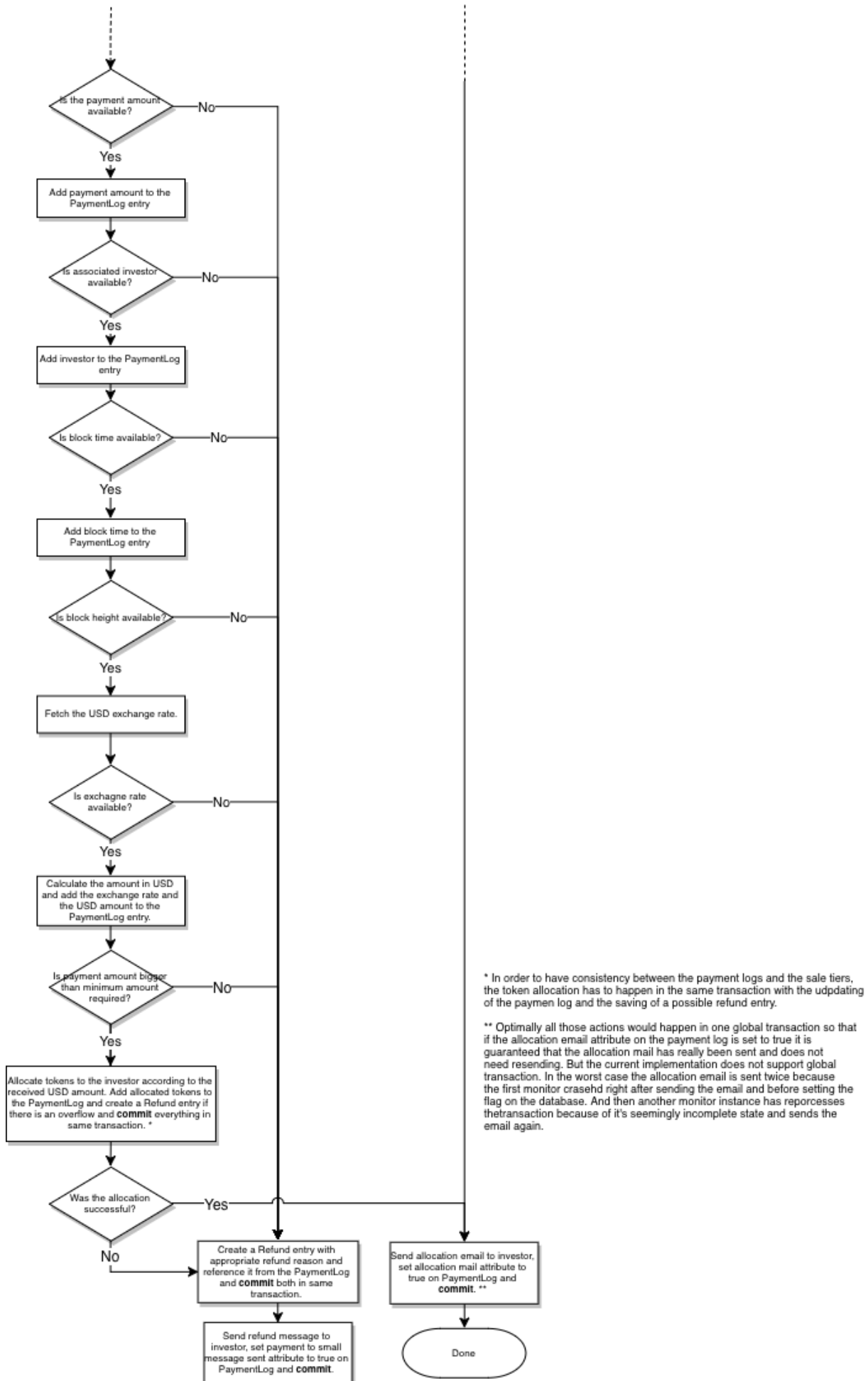
* At this point we don't also check if a refund entry was created because no refund entries are created in the PENDING processing flow. In case there is a refund entry, it has obviously been created in the BUILDING processing flow. Therefore we only have to check if the transaction is in a higher status than PENDING.

** Because the transaction seems to be in an incomplete state after the process exits here, other monitor instances might pick up the transaction for reprocessing (and probably also fail). But that does not lead to any inconsistencies in the data, i.e. is not a problem.

*** Optimally all those actions would happen in one global transaction so that if the confirmation email attribute on the payment log is set to true it is guaranteed that the confirmation mail has really been sent and does not need resending. But the current implementation does not support global transaction. In the worst case the confirmation email is sent twice because the first monitor crashed right after sending the email and before setting the flag on the database. And then another monitor instance has reprocessed the transaction because of its seemingly incomplete state and sends the email again.

C.2 Building transactions flow





Appendix D

Heroku Bash Scripts

D.1 IConator Heroku Deploy

```
#!/bin/bash

RELEASE=false

if [ "$1" = "-r" ]; then
    RELEASE=true
fi

echo "Pulling latest Core Image.."
docker pull iconator/core-master
echo "Tagging Image to Heroku Registry.."
docker tag iconator/core-master registry.heroku.com/iconator-core/web
echo "Pushing Core Image to Heroku.."
docker push registry.heroku.com/iconator-core/web
if [ "$RELEASE" = "true" ]; then
    heroku container:release web -a iconator-core
fi

echo "Pulling latest Email Image.."
docker pull iconator/email-master
echo "Tagging Image to Heroku Registry.."
docker tag iconator/email-master registry.heroku.com/iconator-email/web
echo "Pushing Email Image to Heroku.."
docker push registry.heroku.com/iconator-email/web
if [ "$RELEASE" = "true" ]; then
    heroku container:release web -a iconator-email
fi

echo "Pulling latest Rates Image.."
docker pull iconator/rates-master
echo "Tagging Image to Heroku Registry.."
```

```

docker tag iconator/rates-master registry.heroku.com/iconator-rates/web
echo "Pushing Rates Image to Heroku.."
docker push registry.heroku.com/iconator-rates/web
if [ "$RELEASE" = "true" ]; then
    heroku container:release web -a iconator-rates
fi

echo "Pulling latest Monitor Image.."
docker pull iconator/monitor-master
echo "Tagging Image to Heroku Registry.."
docker tag iconator/monitor-master registry.heroku.com/iconator-monitor/web
echo "Pushing Monitor Image to Heroku.."
docker push registry.heroku.com/iconator-monitor/web
if [ "$RELEASE" = "true" ]; then
    heroku container:release web -a iconator-monitor
fi

echo "Pulling latest Kyc Image.."
docker pull iconator/kyc-master
echo "Tagging Image to Heroku Registry.."
docker tag iconator/kyc-master registry.heroku.com/iconator-kyc/web
echo "Pushing Kyc Image to Heroku.."
docker push registry.heroku.com/iconator-kyc/web
if [ "$RELEASE" = "true" ]; then
    heroku container:release web -a iconator-kyc
fi

```

D.2 IConator Configuration Backup

```

#!/bin/bash
parent_path=$( cd "$(dirname "${BASH_SOURCE[0]}")" ; pwd -P )

cd "$parent_path"

shopt -s nullglob
for i in *.env; do

    echo "reading file $i"

    #remove .env file extension
    filename=${i::-4}

    echo "Creating backup..."
    filename_log=$filename.log

    #create a log
    date >> $filename_log
    heroku config -a $filename >> $filename_log

```

```
        echo "Backup creation Completed"
done
```

D.3 IConator Configuration Update

```
#!/bin/bash
parent_path=$( cd "$(dirname "${BASH_SOURCE[0]}")" ; pwd -P )

cd "$parent_path"

shopt -s nullglob
for i in *.env; do

    echo "reading file $i"

    #remove .env file extension
    filename=${i::-4}

    filename_log=$filename.log

    #Create absolute path by concatenating the parent directory and the filename
    file="$parent_path/$i"

    #Set Config Vars one at a time (each line is config var and value pair)
    while IFS= read -r var
    do
        heroku config:set $var -a $filename
    done < "$file"

done
```


Appendix E

Heroku Configuration Variables

E.1 Core Service Configuration Variables

For the Core Service you need to **set** the following **config vars**:

- **AMQP_URL:**
The unique URL of your Message Queue as defined by the AMQP protocol Service Provider of your choice.
Example value(s):
 - `amqp://user:password@remote.server.com:port//vhost`
- **BITCOIN_NETWORK:**
Offers the option to deploy in the Bitcoin testnet or mainnet.
Example value(s):
 - `testnet` or `mainnet`
- **CORE_COUNTRY_FILTER_DISALLOW:**
Using the 2 letter country code(s) separated by commas you can choose which countries will be excluded from the ICO.
Example value(s):
 - `US,UK` to forbid citizens of the United States and the United Kingdom to participate in the ICO.
- **CORE_COUNTRY_FILTER_ENABLED:**
Enable or disable the 2 letter country code filter.
Example value(s):
 - `true` or `false`
- **CORE_RECAPTCHA_ENABLED:**
Enable or disable the reCAPTCHA service.
Example value(s):
 - `true` or `false`

- **CORE_RECAPTCHA_SECRET_KEY:**
The secret key offered by the reCAPTCHA service.
Example value(s):
 - 6Le pansUAAAAA0QCavMTq23A6oKDRh96YiPJ8_we
- **DATABASE_URL:**
The Heroku specific configuration variable which is automatically generated when you attach a postgresql database to an app. It is the database URL and should have the same value as DATASOURCE_URL.
Example value(s):
 - postgres://{user}:{password}@{hostname}:{port}/{database-name}
- **DATASOURCE_PASSWORD:**
Database password.
Examples value(s):
 - abc123
- **DATASOURCE_URL:**
The database URL, should be the same as DATABASE_URL.
Example value(s):
 - postgres://{user}:{password}@{hostname}:{port}/{database-name}
- **DATASOURCE_USERNAME:**
The username of the account used to connect to the Database.
Examples value(s):
 - user123
- **ICONATOR_AUTH_ACTUATOR_PASSWORD:**
The password used to access the ICONator Spring Boot actuator endpoints.
Example value(s):
 - abc123
- **ICONATOR_AUTH_ACTUATOR_USER:**
The username used to access the ICONator Spring Boot actuator endpoints.
Example value(s):
 - user123
- **ICONATOR_BACKEND_URL:**
The ICONator back-end URL which serves the core API requests.
Example value(s):
 - https://api.example.com
- **ICONATOR_CORE_KEYPAIRS_GEN_AMOUNT:**
Amount of keypairs to be generated.
Example value(s):
 - 50

- **ICONATOR_CORE_KEYPAIRS_GEN_OUTPUT_FILE:**
The file where the generated keys will be stored.
Example value(s):
 - `/ICOnator/keypairs.keys`
- **ICONATOR_CORS_URLS:**
Cross-Origin Resource Sharing (CORS) URLs for the ICOnator engine.
Example value(s):
 - `https://api.example.com,https://dev.example.com`
- **ICONATOR_FRONTEND_URL:**
Front-end ICO url.
Example value(s):
 - `https://ico.mytoken.io`
- **ICONATOR_FRONTEND_WALLET_PATH:**
The relative wallet path in the front-end.
Example value(s):
 - `/confirm/`
- **JPA_GENERATE_DDL_ENABLED:**
Switches the Java Persistence API (JPA) feature of the Data Definition Language (DDL) on or off.
Example, value(s):
 - `true` or `false`
- **JPA_HIBERNATE_DDL_AUTO:**
Specifies a more fine-grained control which influences how the schema tool management will manipulate the database schema at startup.
Example value(s):
 - `create, create-drop, validate, and update`
- **LOGSTASH_ADDRESS:**
URL of the Logstash service for the service to sent the logs to.
Example value(s):
 - `https://logstash.example.com:5000`
- **SPRING_PROFILES_ACTIVE:**
Which Spring profile will be active.
Example value(s):
 - `default, dev, prod`

E.2 Email Service Configuration Variables

For the `Email Service` you need to set the following `config vars`:

- **AMQP_URL:**
The unique URL of your Message Queue as defined by the AMQP protocol Service Provider of your choice.
Example value(s):
 - `amqp://user:password@remote.server.com:port//vhost`
- **BITCOIN_NETWORK:**
Offers the option to deploy in the Bitcoin testnet or mainnet.
Example value(s):
 - `testnet` or `mainnet`
- **DATABASE_URL:**
Heroku specific configuration variable which is automatically generated when you attach a postgresql database to an app.
Example value(s):
 - `postgres://{user}:{password}@{hostname}:{port}/{database-name}`
- **EMAIL_ADMIN_ADDRESS:**
The administrator's email address.
Example value(s):
 - `admin@mytoken.io`
- **EMAIL_AUTH_ENABLED:**
Enable or disable email authentication.
Example value(s):
 - `true` or `false`
- **EMAIL_BCC_CONFIRMATION_ENABLED:**
Enable or disable bcc confirmation email.
Example value(s):
 - `true` or `false`
- **EMAIL_BCC_SUMMARY_ENABLED:**
Enable or disable bcc summary.
Example value(s):
 - `true` or `false`
- **EMAIL_CONFIRMATION_SUBJECT:**
Subject of the confirmation email.
Example value(s):
 - `MyToken.io - Token Sale`

- **EMAIL_DEBUG_ENABLED:**
Enable or disable debug.
Example value(s):
 - `true` or `false`
- **EMAIL_ENABLED:**
Enable or disable the email service.
Example value(s):
 - `true` or `false`
- **EMAIL_FOOTER_ENTITY_NAME:**
Organization name in the footer section of the email.
Example value(s):
 - `MyToken Solutions GmbH`
- **EMAIL_FOOTER_YEAR:**
Year in the footer section of the email.
Example value(s):
 - `2018`
- **EMAIL_FUNDS_RECEIVED_SUBJECT:**
Subject of the funds received email.
Example value(s):
 - `MyToken.io - Funds Received`
- **EMAIL_HOST:**
SMTP server used by the email service.
Example value(s):
 - `smtp.mytoken.io`
- **EMAIL_LOGO_CONTENT_TYPE:**
Type of the logo content.
Example value(s):
 - `image/png`
- **EMAIL_LOGO_HEIGHT:**
Height of the email logo in pixels.
Example value(s):
 - `100`
- **EMAIL_LOGO_URL:**
URL of the email logo.
Example value(s):
 - `https://mytoken.io/static/logos/mytoken.png`

- **EMAIL_LOGO_WIDTH:**
Width of the email logo in pixels.
Example value(s):
 - 300
- **EMAIL_PASSWORD:**
Mail service password.
Height of the email logo in pixels.
Example value(s):
 - abc123
- **EMAIL_PORT:**
Port of the mail service.
Height of the email logo in pixels.
Example value(s):
 - 587
- **EMAIL_PROTOCOL:**
Mail service protocol. Height of the email logo in pixels.
Example value(s):
 - smtp
- **EMAIL_SEND_FROM_ADDRESS:**
Sender's email address.
Example value(s):
 - ico@mytoken.io
- **EMAIL_START_TLS_ENABLED:**
Enable or disable TLS in mail service.
Example value(s):
 - true or false
- **EMAIL_SUMMARY_SUBJECT:**
Subject of the summary email.
Example value(s):
 - MyToken.io - Summary of your Investment
- **EMAIL_TOKEN_SALE_NAME:**
Name of the token sale.
Example value(s):
 - MyToken
- **EMAIL_TOKENS_ALLOCATED_SUBJECT:**
Subject of the tokens allocated email.
Example value(s):
 - MyToken.io: Your Tokens

- **EMAIL_TOKEN_SYMBOL:**
Token ticker symbol.
Example value(s):
 - `TKN`
- **EMAIL_TRANSACTION_RECEIVED_SUBJECT:**
Subject of the transaction received email.
Example value(s):
 - `myToken.io: Transaction Detected`
- **EMAIL_TRUST:**
Which emails are trusted.
Example value(s):
 - `*`
- **EMAIL_USERNAME:**
Mail service username.
Example value(s):
 - `user123`
- **ICONATOR_AUTH_ACTUATOR_PASSWORD:**
The password used to access the IConator Spring Boot actuator endpoints.
Example value(s):
 - `abc123`
- **ICONATOR_AUTH_ACTUATOR_USER:**
The username used to access the IConator Spring Boot actuator endpoints.
Example value(s):
 - `user123`
- **LOGSTASH_ADDRESS:**
URL of the Logstash service for the service to sent the logs to.
Example value(s):
 - `https://logstash.example.com:5000`
- **MAILGUN_SMTP_SERVER:**
Heroku specific config variable for the Mailgun Add-on. Specifies the SMTP server.
Example value(s):
 - `smtp.mytoken.io`
- **SPRING_PROFILES_ACTIVE:**
Which Spring profile will be active.
Example value(s):
 - `default, dev, prod`

E.3 Rates Service Configuration Variables

For the `Rates Service` you need to set the following `config vars`:

- **AMQP_URL:**
The unique URL of your Message Queue as defined by the AMQP protocol Service Provider of your choice.
Example value(s):
 - `amqp://user:password@remote.server.com:port//vhost`
- **DATABASE_URL:**
Heroku specific configuration variable which is automatically generated when you attach a postgresql database to an app.
Example value(s):
 - `postgres://{user}:{password}@{hostname}:{port}/{database-name}`
- **DATASOURCE_PASSWORD:**
Database password.
Examples value(s):
 - `abc123`
- **DATASOURCE_URL:**
The database URL, should be the same as `DATABASE_URL`.
Example value(s):
 - `postgres://{user}:{password}@{hostname}:{port}/{database-name}`
- **DATASOURCE_USERNAME:**
The username of the account used to connect to the Database.
Examples value(s):
 - `user123`
- **ICONATOR_AUTH_ACTUATOR_PASSWORD:**
The password used to access the ICONator Spring Boot actuator endpoints.
Example value(s):
 - `abc123`
- **ICONATOR_AUTH_ACTUATOR_USER:**
The username used to access the ICONator Spring Boot actuator endpoints.
Example value(s):
 - `user123`
- **ICONATOR_CORS_URLS:**
Cross-Origin Resource Sharing (CORS) URLs for the ICONator engine.
Example value(s):
 - `https://api.example.com,https://dev.example.com`

- **JPA_GENERATE_DDL_ENABLED:**
Switches the Java Persistence API (JPA) feature of the Data Definition Language (DDL) on or off.
Example, value(s):
 - `true` or `false`
- **JPA_HIBERNATE_DDL_AUTO:**
Specifies a more fine-grained control which influences how the schema tool management will manipulate the database schema at startup.
Example value(s):
 - `create`, `create-drop`, `validate`, and `update`
- **LOGSTASH_ADDRESS:**
URL of the Logstash service for the service to sent the logs to.
Example value(s):
 - `https://logstash.example.com:5000`
- **RATES_BLOCKNR_FALLBACK_API_BITCOIN_OLDER_THAN:**
Block number from which the rates fallback API will begin fetching Bitcoin rates.
Example value(s):
 - `7200000`
- **RATES_BLOCKNR_FALLBACK_API_ENABLED:**
Enable or disable the rates fallback API.
Example, value(s):
 - `true` or `false`
- **RATES_BLOCKNR_FALLBACK_API_ETHEREUM_OLDER_THAN:**
Block number from which the rates fallback API will begin fetching Ethereum rates.
Example value(s):
 - `1800000`
- **RATES_CURRENCIES_CRYPTOCURRENCY_ENABLED:**
Which cryptocurrencies are enabled for the rates service.
Example value(s):
 - `BTC,ETH` for Bitcoin and Ethereum
- **RATES_CURRENCIES_FIAT_BASE:**
The FIAT base currency for the cryptocurrency rates.
Example value(s):
 - `USD`
- **RATES_EXCHANGES_ENABLED:**
Which exchanges are enabled to fetch rates from.
Example value(s):
 - `KRAKEN,BITFINEX,BITSTAMP,COINMARKETCAP,GDAX`

- **RATES_FETCH_HISTORICAL_ENABLED:**
Enable or disable historical fetching of rates if there is a time period missing in the database.
Example value(s):
 - `true` or `false`
- **RATES_FETCH_PERIODIC_ENABLED:**
Enable or disable periodical fetching of rates.
Example value(s):
 - `true` or `false`
- **RATES_FETCH_PERIODIC_INTERVAL:**
Interval of the periodical rates fetch in milliseconds.
Example value(s):
 - `60000`
- **RATES_FETCH_RANGE_BETWEEN_THRESHOLD_MAX:**
Maximum waiting time between rates fetching retries.
Example value(s):
 - `7200000`
- **RATES_FETCH_RANGE_BETWEEN_THRESHOLD_MIN:**
Minimum waiting time between rates fetching retries.
Example value(s):
 - `7200000`
- **RATES_OUTLIERS_STDDEV_LOWER_BOUND:**
Lower bound for rates outliers.
Example value(s):
 - `2`
- **RATES_OUTLIERS_STDDEV_UPPER_BOUND:**
Upper bound for rates outliers.
Example value(s):
 - `2`
- **RATES_RETRY_ATTEMPTS_MAX:**
Maximum retry attempts to fetch the rates.
Example value(s):
 - `3`
- **RATES_RETRY_BETWEEN_ATTEMPTS_MAX:**
Maximum interval between retry attempts in ms.
Example value(s):
 - `1000`

- **RATES_RETRY_BETWEEN_ATTEMPTS_MIN:**
Minimum interval between retry attempts in ms.
Example value(s):
 - 100
- **RATES_USER_AGENT:**
Rates user agent name.
Example value(s):
 - Mycoin-Rates-Client
- **SPRING_DATASOURCE_MAX_ACTIVE:**
Maximum pool size for tomcat-jdbc connections.
 - 3
- **SPRING_PROFILES_ACTIVE:**
Which Spring profile will be active.
Example value(s):
 - default, dev, prod

E.4 Monitor Service Configuration Variables

For the Monitor Service you need to set the following config vars:

- **AMQP_URL:**
The unique URL of your Message Queue as defined by the AMQP protocol Service Provider of your choice.
Example value(s):
 - `amqp://user:password@remote.server.com:port/vhost`
- **BITCOIN_NETWORK:**
Offers the option to deploy in the Bitcoin testnet or mainnet.
Example value(s):
 - `testnet` or `mainnet`
- **DATABASE_URL:**
The Heroku specific configuration variable which is automatically generated when you attach a postgresql database to an app. It is the database URL and should have the same value as `DATASOURCE_URL`.
Example value(s):
 - `postgres://{user}:{password}@{hostname}:{port}/{database-name}`
- **DATASOURCE_PASSWORD:**
Database password.
Examples value(s):

– `abc123`

- **DATASOURCE_URL:**

The database URL, should be the same as `DATABASE_URL`.

Example value(s):

– `postgres://{user}:{password}@{hostname}:{port}/{database-name}`

- **DATASOURCE_USERNAME:**

The username of the account used to connect to the Database.

Examples value(s):

– `user123`

- **ICONATOR_AUTH_ACTUATOR_PASSWORD:**

The password used to access the ICONator Spring Boot actuator endpoints.

Example value(s):

– `abc123`

- **ICONATOR_AUTH_ACTUATOR_USER:**

The username used to access the ICONator Spring Boot actuator endpoints.

Example value(s):

– `user123`

- **JPA_GENERATE_DDL_ENABLED:**

Switches the Java Persistence API (JPA) feature of the Data Definition Language (DDL) on or off.

Example, value(s):

– `true` or `false`

- **JPA_HIBERNATE_DDL_AUTO:**

Specifies a more fine-grained control which influences how the schema tool management will manipulate the database schema at startup.

Example value(s):

– `create`, `create-drop`, `validate`, and `update`

- **LOGSTASH_ADDRESS:**

URL of the Logstash service for the service to sent the logs to.

Example value(s):

– `https://logstash.example.com:5000`

- **MONITOR_ATOMIC_TOKEN_UNIT_FACTOR:**

This defines the atomic unit factor of the token. The value set here is not the actual factor itself but the exponent to the base 10. For example Ethereum has a atomic unit factor of 10^{18} (Wei)

Example value(s):

– `18`

- **MONITOR_BITCOIN_CONFIRMATION_BLOCKDEPTH:**
Number of Bitcoin blocks which need to cover a transaction's block to consider the transaction confirmed.
Example value(s):
 - 3
- **MONITOR_BITCOIN_NODE_ENABLED:**
Enable or disable the Bitcoin node.
Example value(s):
 - `true` or `false`
- **MONITOR_BITCOIN_NODE_FASTCATCHUP:**
Date from which to do a fast catchup in ISO-8601 format.
Example value(s):
 - 2018-09-01T00:00:00Z
- **MONITOR_ETHEREUM_CONFIRMATION_BLOCKDEPTH:**
Number of Ethereum blocks which need to cover a transaction's block to consider the transaction confirmed.
Example value(s):
 - 5
- **MONITOR_ETHEREUM_NODE_ENABLED:**
Enable or disable the Ethereum node.
Example value(s):
 - `true` or `false`
- **MONITOR_ETHEREUM_NODE_START_BLOCK:**
Starting block from which Ethereum transactions are monitored. The monitor will ignore all previous blocks.
Example value(s):
 - 8968639
- **MONITOR_ETHEREUM_NODE_URL:**
Url of the Ethereum full node to connect to.
Example value(s):
 - `http://ip:port`
- **MONITOR_FIAT_BASE_PAYMENT_MIN:**
Minimum payment in the fiat base currency. Payments below this amount will not lead to a token allocation but to a refund entry.
Example value(s):
 - 10
- **MONITOR_FIAT_BASE_PER_TOKEN:**
Token value in the fiat base currency.
Example value(s):

– 0.1 (USD per Token)

- **MONITOR_TOTAL_TOKEN_AMOUNT:**

Total tokens amount. This represents the token limit of the ICO. It is specified in the main unit of the token and not the atomic unit.

Example value(s):

– 1000000

- **MONITOR_TRANSACTION_PROCESSING_TIME:**

This value specifies the interval, starting at the last modified date, in which a transaction is considered as being currently processed by a monitor instance. It is only used in case multiple monitor instances run concurrently to avoid conflicting manipulation between the instances on the same transaction (see subsection 4.3.6). 10 seconds is recommended. The value is given in milliseconds. Example value(s):

– 10000

- **SPRING_PROFILES_ACTIVE:**

Which Spring profile will be active.

Example value(s):

– default, dev, prod

E.5 KYC Service Configuration Variables

For the KYC Service you need to set the following config vars:

- **AMQP_URL:**

The unique URL of your Message Queue as defined by the AMQP protocol Service Provider of your choice.

Example value(s):

– amqp://user:password@remote.server.com:port//vhost

- **DATABASE_URL:**

Heroku specific configuration variable which is automatically generated when you attach a postgresql database to an app.

Example value(s):

– postgres://{user}:{password}@{hostname}:{port}/{database-name}

- **DATASOURCE_PASSWORD:**

Database password.

Examples value(s):

– abc123

- **DATASOURCE_URL:**

The database URL, should be the same as DATABASE_URL.

Example value(s):

- `postgres://{user}:{password}@{hostname}:{port}/{database-name}`
- **DATASOURCE_USERNAME:**
The username of the account used to connect to the Database.
Examples value(s):
 - `user123`
- **ICONATOR_AUTH_ACTUATOR_PASSWORD:**
The password used to access the IConator Spring Boot actuator endpoints.
Example value(s):
 - `abc123`
- **ICONATOR_AUTH_ACTUATOR_USER:**
The username used to access the IConator Spring Boot actuator endpoints.
Example value(s):
 - `user123`
- **ICONATOR_KYC_API_KEY:**
KYC provider api key.
Example value(s):
 - `abcdef12345`
- **ICONATOR_KYC_COMPANY_ID:**
KYC provider company identifier.
Example value(s):
 - `test-id`
- **ICONATOR_KYC_HOST:**
KYC provider host Url.
Example value(s):
 - `http://kyc.example.com`
- **JPA_GENERATE_DDL_ENABLED:**
Switches the Java Persistence API (JPA) feature of the Data Definition Language (DDL) on or off.
Example, value(s):
 - `true` or `false`
- **JPA_HIBERNATE_DDL_AUTO:**
Specifies a more fine-grained control which influences how the schema tool management will manipulate the database schema at startup.
Example value(s):
 - `create`, `create-drop`, `validate`, and `update`
- **KYC_CLIENT_RETRY_ATTEMPTS_MAX:**
Maximum kyc client retry attempts.
Example value(s):

– 3

- **KYC_CLIENT_RETRY_BETWEEN_ATTEMPTS_MAX:**

Maximum kyc client retry attempts in milliseconds.

Example value(s):

– 1000

- **KYC_CLIENT_RETRY_BETWEEN_ATTEMPTS_MIN:**

Minimum kyc client retry attempts in milliseconds.

Example value(s):

– 100

- **LOGSTASH_ADDRESS:**

URL of the Logstash service for the service to sent the logs to.

Example value(s):

– `https://logstash.example.com:5000`

- **SPRING_PROFILES_ACTIVE:**

Which Spring profile will be active.

Example value(s):

– `default, dev, prod`