

University of Zurich[UZH]

MASTER PROJECT – Communication Systems Group, Prof. Dr. Burkhard Stiller

# EUREKA Platform - Final report

*Severin Wullschleger, Lucas Pelloni, Andreas Schaufelbühl*
*Zurich, Switzerland*
*Student ID: 13-715-081, 13-722-038, 12-918-843*

Supervisor: Sina Rafati
Date of Submission: 20. April 2019

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Project
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

# Contents


**1 Introduction** **1**

1.1 Motivation . . . . . 1

1.2 Description of Work . . . . . 3

1.3 Thesis Outline . . . . . 3

1.4 EUREKA Blockchain Solutions GmbH . . . . . 3

**2 Related Work** **5**

2.1 Comparison of Related Work . . . . . 6

**3 Design** **7**

3.1 Tokenomics . . . . . 7

3.1.1 Token properties . . . . . 7

3.1.2 Token flow . . . . . 8

3.2 The REWARD system . . . . . 9

**4 Implementation** **11**

4.1 The EUREKA peer-review and publishing process . . . . . 12

4.2 Platform architecture overview . . . . . 15

4.3 Front-End . . . . . 16

4.3.1 General information and front-end concepts . . . . . 16

4.3.2 Working product (Screenshots) . . . . . 28

4.4 Smart contracts . . . . . 43

4.4.1 Token contract . . . . . 43

4.4.2 Journal contracts (platform contract) . . . 45
4.4.3 Smart contract interactions . . . 50
4.4.4 Web3 contract methods interface . . . 58
4.5 Back-End . . . 59
4.5.1 Technical stack . . . 59
4.5.2 Security provisions . . . 60
4.5.3 Business logic . . . 61
4.5.4 Structure and responsibilities . . . 63
4.5.5 REST interface . . . 72
4.6 Interface between front-end and smart contracts . . . 74
4.6.1 React, Web3 and MetaMask . . . 74
4.6.2 Web3 contract methods interface . . . 74
4.6.3 Sending transactions and resolving promiEvents . . . 74
4.7 Interface between smart contracts and back-end . . . 77
4.7.1 Smart contract deployment . . . 77
4.7.2 Connection to Ethereum node . . . 79
4.7.3 Smart contract mirror - contract event listener . . . 82
4.7.4 Timeout handler - time-based contract service . . . 83
4.8 Interaction between all components . . . 86

**5 Evaluation** **89**

5.1 Integration Tests . . . 89
5.2 System Usability Scale (SUS) . . . 90
5.2.1 Results . . . 95

**6 Summary and Conclusions** **97**

**7 Installation Guide** **99**

7.1 Prerequisites . . . . . 99

7.2 Platform Setup . . . . . 99

7.3 Start application locally with local Ganache Network . . . . . 101

7.4 Application interacting with kovan network . . . . . 102

**8 How to modify the platform** **105**

8.1 Initial situation . . . . . 105

8.2 Step 1: Modify front-end by adding input field . . . . . 106

8.3 Step 2: Add the variable to the database schema . . . . . 107

8.4 Step 3: Adjust back-end REST API and corresponding calls from front-end 107

8.5 Step 4: Adjust article data fetching in front-end . . . . . 108

8.6 Step 5: Adjust the affected structs in the EUREKA platform contract . . . 108

8.7 Step 6: Adjust the affected smart contract methods . . . . . 109

8.8 Step 7: Adjust the smart contract call . . . . . 109

8.8.1 Adjust method hash representation . . . . . 109

8.8.2 Adjust the input data state and the parameters of the called method 110

8.8.3 Change byte code representation of method parameters . . . . . 110

8.9 Step 8: Additional smart contract adjustments and change event return values . . . . . 112

8.10 Step 9: Adjust the platform contract monitor . . . . . 112

**Bibliography** **113**

**List of Figures** **119**

# Chapter 1

# Introduction

## 1.1 Motivation

Out casting, sharing, and publishing the collected knowledge, experiences, and out puts of scientific works, regardless of being empirical or pure theoretical, has always been one of the key assets of human evolution. Indeed, from thousands of years ago, where humans started to draw images on the cave walls, until today, where computers and Internet have opened new paradigms to human's life, sharing the knowledge and experience have been constantly the main appliance for keeping and improving the knowledge, believes, experiences, and results of the investigations and explorations.
It would not be possible without learning from each other, and in this millennium, one of the main groups of mechanisms in sharing the knowledge especially in academic life is electrical Scientific Publications Systems (SPS). Electrical publications provide access to millions of scientific documents from journals, books, protocols, and reference works [35] [37] [36].

Nowadays, researcher coming from the scientific and medical field assemble research data to form a narrative to then submit it to a scientific journal. The submitted academic work is then carefully read by educated people in the same field. This is also known as the peer-review process. Once the peer-review assessment is completed, the research is ready to be published in academic journals.
Even though the publication systems seem to be successful, current studies show that these systems face several challenges and deficits. Indeed, most of the time when a paper is submitted, revisions are requested and the authors start the process of submitting to another journal all over again, making this process inefficient.
Since the number of reputable publishers is limited, the process of peer-reviewing can last months (in some cases even years) and authors do not have the warranty that their discoveries at the end will be published. In fact, the publication rate varies amongst journals, but most renowned journals typically have very low publication acceptance rates (often less than 10%) [39].

Furthermore in the current system, the number of people involved in the decision-making

process for publishing academic work are very little compared to the number of people who provides the research and the discoveries. This makes the publishing industry an oligarchic structure, dominated by a very small number of players.
In addition to that, well-known publishers put barriers to access knowledge that has been reviewed for free by researchers all over the world.
This leads to most of the research still remains inaccessible to most people on the planet [39]. We identified three main problem that Academia is facing today, such as :

1. **Reproducibility Crisis**
   Usually, researchers often are in a situation where they are unable to reproduce the work of other researchers. Current scientific publishing culture is not supporting unsuccessful trials to be published. For this reason, it is intuitively common to experience the same failure by different researchers after dedicating considerable amount of time, energy, and cost as they could not attain any information with the current publishing systems about that previous and the same unsuccessful experience of researchers in the same field. Here the problem might be the low probability of failed experiences to be published in current SPS which basically leaves no incentive for scientist to bare the effort for documenting and publishing the steps and the failed results.

2. **No fair credit for contribution**
   Observations on the publishing process in current SPS, indicate that the contribution of reviewers and authors are neglected. On the one hand, unfortunately within the current SPS publishers expect the experts of a field to dedicate a considerable amount of time for reviewing an article with no reward. On the other hand, publishers may ask authors to pay for publishing their documented work and results, and universities are asked to pay subscription fees for accessing the scientific publishing articles. Surprisingly, who have done the real work need to pay. Even more surprising is that most of the reviewers are not being credited properly (i.e., not paid) for the time and effort they invest for evaluation of an article by current SPS.

3. **Long delays in publishing** According to a survey of 1500 scientists conducted by the science journal Nature [40], more than 70% have been unable to reproduce another scientist's experiments. Even more worrying, over 50% of those surveyed were unable to reproduce their own experiments. As a result, much time and billions of dollars are wasted pursuing drug candidates which have already been found in unpublished research not to be effective. This is what is known as the reproducibility's crisis. It is estimated that in the US alone, pharmaceutical companies waste $28 billion annually on irreproducible life sciences research. The top ten biopharma companies spent $63 billion on Research Development in 2016.

To overcome these problems, we present EUREKA [1], the next-generation open science platform powered by blockchain which aims at democratising science through decentralisation and transparency.

## 1.2 Description of Work

This thesis describes the implementation of the EUREKA platform. The platform has a novel incentive model within the publishing process to ensure quality research. Authors and academic institutions will be able to publish open-access research while reviewers will be directly rewarded via a SC and micropayments. Linked authors will be micro-rewarded as well for their indirect contribution to a paper. By using a blockchain-based system, the EUREKA platform provides transparency and timestamping of scientific discoveries, giving the authors immediate ownership rights of their discoveries. The potential use cases for EUREKA are substantial and global.

## 1.3 Thesis Outline

*Chapter 2* presents the recent literature and other companies ideas in the area of interest for this work, *i.e.,* blockchain-based ecosystems in the field of science publishing. *Chapter 3* describes the EUREKA REWARD system in terms of incentive mechanisms for all stakeholders involved in the publishing process. *Chapter 4* explains the implementation of the EUREKA platform. At the beginning of this Chapter, the EUREKA peer-review and publishing process is described. Afterwards, the architecture overview of the different components and their inter-connections is presented. *Chapter 5* describes the evaluation of the EUREKA platform. Indeed, the results of different Integrations Tests and System Usability Scale experiment is presented in this Chapter. *Chapter 6* presents the the summary and conclusions of this thesis. *Chapter 7* lists the required steps to install the system and *Chapter 8* closes the thesis by explaining how to introduce a new module in the platform.

## 1.4 EUREKA Blockchain Solutions GmbH

As we develop the platform in cooperation with the company EKA Blockchain Solutions GmbH [1] we introduce here the company in short. The goal of EKA Blockchain Solutions GmbH is to become a provider of technology to publishers, both directly and indirectly. It aims to create a more efficient and trustworthy publishing process by a blockchain-based technology. For that purpose it introduces the EUREKA Platform for time-stamping discoveries, reward reviewers and push decentralized publishing.
Evermore the company targets to tackle the fundamental issues in the current scientific research systems, like rewarding the multitude of unpaid work done by paper reviews and editors. The main goal is to become an technology platform, which provides infrastructure and business logic to publishing houses, such as the in-house example Science Matters [2]. With the use of the blockchain technology EKA Blockchain Solutions GmbH will offer a fair, transparent and safe publishing platform.

# Chapter 2

# Related Work

This chapter presents similar and related proposed works in the scientific publishing area. All of them tackle the goal of revolutionizing a specific aspect of the scientific publishing process. With a table we compare these different works and put them in contrast to each other based on meaningful categories.
Orvium [49] aims to create a open-science platform. It integrates blockchain technology into the scholarly publications' life cycles. Collaboration, peer review, payment of copyright licenses and research data sharing will all generate exchanges of the in-house crypto token. According to the published white paper [49] the proposed system is developed and live by the end of 2019. Concise characteristics of the platform is the possibility of not just owning copyrights within the system but also transfer them to other authors. Furthermore it includes a instantaneous proof-of-existence, where manuscripts are publicly available from the moment they are submitted to the platform. Peer reviews are transparent and publicly accessible and used as a mechanism to determine the validity and correctness of research through community-controlled examination.
Creating and customizing its own journal is a further key feature of the platform. These journals follow a Decentralized Autonomous Organization (DAO) [53] approach. Each user of the platform will be able to set up his own Decentralized Autonomous Jornal (DAJ), where he can define its governance rules.
Similarly, "Blockchain for Science" [52] is an organization which wants to create a new living knowledge network. With the blockchain technology as base-tool, they want to connect different research devices with each other and distribute research money anonymously. In contrast to Orvium they focus on collaborative scientific work by bringing people with similar field of interest together. This think-thank composes of many people trying to find new application areas for blockchain in terms of scientific work. Their core tool, the Open Living DOC [54] is sort of a universal and shared whitepaper holding the ideas, opinions and approaches for these application areas, created by a team of multiple people.
In addition, ScienceRoot [51] wants to create a distributed social media platform for science, through a blockchain-based ecosystem. They create a common transparent platform which will be used for collaborating, sharing ideas and publishing their work on it. They integrate blockchain technology combined with the Interplanetary File System (IPFS) [55] as base for their platform.

ScieneRoot will include a Marketplace wit a set of shared repositories. This Marketplace can be viewed as a database for science. Furthermore it will include a portal for funding and job offering. The portal will be accessible publicly for funding organizations of scientific projects.
Considering funding research projects, DEIP [56] aims to establish a distributed ecosystem for scientifical work. DEIP aims to reward every research project fairly regardless of its final result if the correspondent experts in that field will acknowledge the significance and impact of the work. Further it promotes cooperation and collaboration netween different research groups, by assessing all researches of their significance and quality
DEIP lays its focus on funding mechanisms for diverse scientific projects For that they will create the DEIP protocol which handles the distribution across scientific activities depending on the total number of researches at the platform [56]. The ecosystem will entirely be driven by the scientific community. Therefore a main long-term aspect for DEIP is to stay absolutely free of charge for its user, but incentives them by the means of a protocol and its underlying economics. It promotes open-science as well.

At last there is F1000research [65], a centralized publishing platform for scientific work. Compared to the others it does not rely on a decentralized approach with the use of blockchain. But it also proposes a open-science and instant publication of articles. They include transparent refereeing and inclusion of all source data. As all the other presented projects it will also promote open-science. For that all discoveries will be accessible by anybody.

## 2.1 Comparison of Related Work

Most of the above claimed functionalities and features are still under progress and are not yet developed and established. Besides sharing a common vision of decentralization through blockchain technology all the mentioned works here cover common, but also distinct use cases with resepct to the level of centralization, employed incentive and rewarding mechanism and stakeholder goals. Table 2.1 compares the related work in different categories..

Table 2.1: Comparision Table of Related Works

| | Orvium | Blockchain for Science | Science Root | DEIP | F1000research |
|---|---|---|---|---|---|
| **Open Science** | Yes | Yes | Yes | Yes | Yes |
| **DAO** | Yes | No | Yes | Yes | No |
| **Own token economy** | Yes, Orvium Token | No | Yes, Science Token | Yes, DEIP Token | No |
| **Base Blockchain Technology** | Ethereum | - | Ethereum | Graphene Framework | - |

# Chapter 3

# Design

In this chapter the general design concepts are introduced. First it presents the tokenomics, while also explaining the different token properties existing. Further we have a look into the details of the token flow within our system. Secondly the incentive system named 'REWARD' of the system is explained and all the different involved roles of it are presented.

## 3.1 Tokenomics

### 3.1.1 Token properties

Crypto-Tokens are well-known assets in the crypto space and are usually identified as a subset of the huge cryptocurrency family. While a cryptocurrency is a standard currency for making or receiving payments on a blockchain, crypto tokens are a special kind of virtual representation of a particular asset or utility, that reside on their own blockchains [71], [72].

Usually, tokens can be classified into two main categories:

**1. Utility Tokens**
As the word itself states, utility tokens represent a sort of "digital coupon" which can be used for accessing a particular service or product in a crypto ecosystem (*e.g.,* EUREKA). Usually, users operating with such tokens will be rewarded for their contribution in that particular system. In this sense, utility tokens empower the participants therein.

**2. Security Tokens**
Security tokens can be viewed as a digital company's share and in the norm are more designed as an investment. Indeed, they give the holder an ownership rights (normally, also dividends are included and paid out). However, security tokens imply more strict federal regulations, limiting investors participation and the trading possibility.

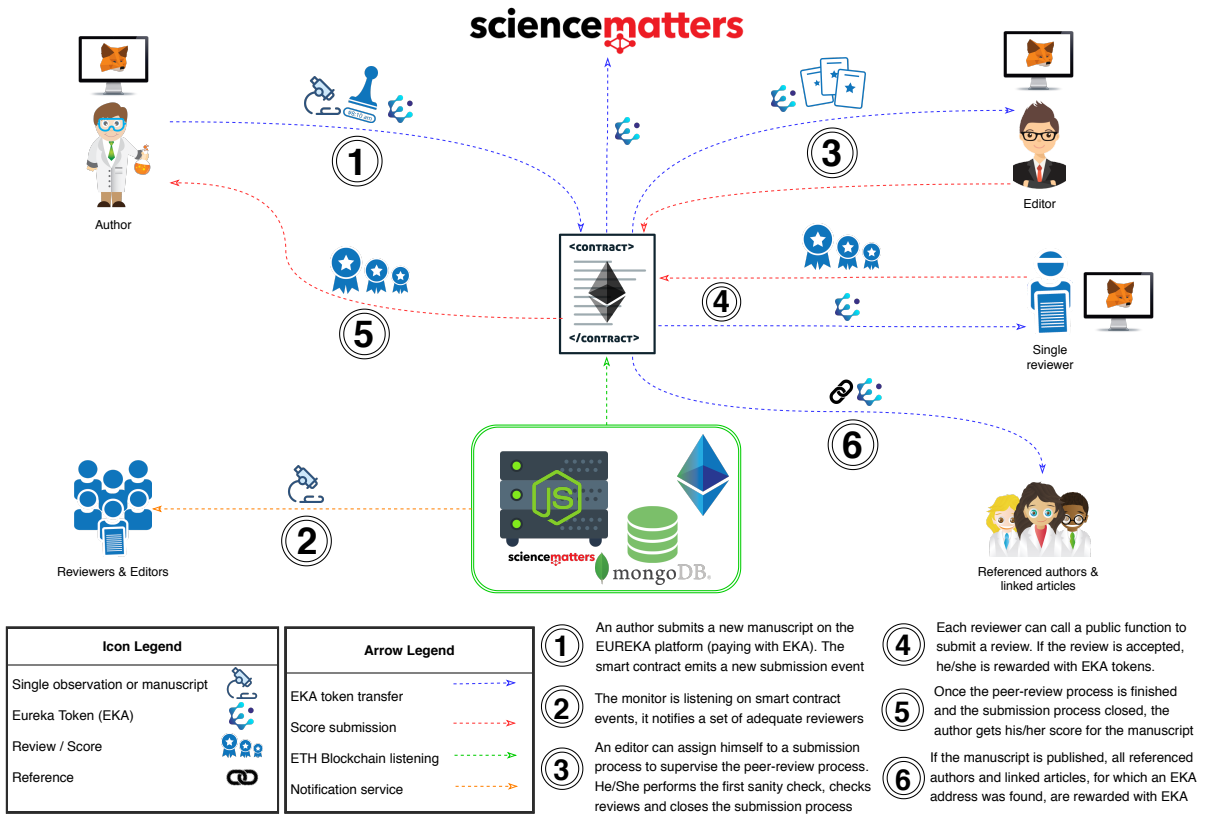



Figure 3.1: The general EUREKA Transaction Flow

One of the well-documented challenges in the current publishing systems is the absence of fair contributions for those stakeholders who were involved during the editorial assessment and peer-reviewing process. Indeed, authors, editors, and reviewers currently get almost no direct compensation for the scholarly work that they perform in publishing and reviewing. The high cost to publish peer-reviewed research means researchers have a barrier to publish and share knowledge with society and the scientific community. To tackle these issues, EUREKA exploits blockchain technology to create a new incentive mechanism in the academic industry: the REWARD system and the EUREKA utility token.

### 3.1.2 Token flow

Figure 3.1 shows the illustrated token flow which will be described in the following paragraphs.

The EKA utility token starts its flow when the first stakeholder, the author, feels like submitting a manuscript. The authors sends a transaction with the hash of the manuscript along with the metadata (e.g. Ethereum addresses of co-authors, referenced authors or hashes of linked articles) to the smart contract. There, the manuscript is timestamped and a new submission process is initialized. The smart contract monitor is listening to the smart contract and reacts to changes: It will notify reviewers and editors about a new manuscript uploaded.

A handling editor can assign himself to a open submission process. He/she will supervise the whole process and perform the first sanity check. Both are actions which will be sent to the smart contract in form of a transaction (see figure 3.1, red arrow at number 3). In contrast to the first step, for these transaction no EKA transfer is needed in particular. The transactions costs can be payed in Ether but there will be a service provide which allows the stakeholders to pay the transaction costs in EKA tokens if they want.

Interested reviewers can send transaction with the review score (and a review hash which represents the whole review) to the smart contract. Again, for this transaction no EKA tokens flow necessarily (red arrow at number 4).

If enough reviews are submitted the editor can accept or decline the submitted manuscript. In both cases the author will receive their review score and all stakeholders are getting EKA tokens transfered as reward for their work. In addition to the handling editor and the reviewers, reward EKA tokens are also transferred to the ScienceMatters foundation, and to referenced authors and authors of linked articles.

## 3.2 The REWARD system

EUREKA presents the so called REWARD system. It is described as follows [39]:

**1. Revelation**
The time-stamping of single observations as an incentive since this gives the author immediate ownership rights, and ensures scientists' and researchers' discoveries are tamper-proof.

**2. Evaluation**
Eureka will make use of crowd-sourced reviewers to get consensus-oriented evaluations of work, instead of being restricted to one or two reviewers, as is common practice. In doing so, each peer in the world could be part of the peer-reviewing process of a certain manuscript and thus get rewarded for his/her work with EKA tokens.

**3. Worth Assignment**
The Eureka platform's crowd-sourced scoring system will provide researchers as well as publishers a new metric that can be used to evaluate articles. Since the ratings for scientific work are stored on the blockchain, the scores can be transferred (*e.g.,* to another journal) and the peer-reviewing process must not be performed twice.

**4. Award**
Referenced authors, editors and reviewers will be rewarded for their operations within the Eureka ecosystem with EKA tokens coming from the amount paid for the submission. Referenced authors, will get tokens once an editor approved a specific manuscript, reviewers will be rewarded when their submitted reviews have been accepted by the editor and/or by second-reviewers. The editor's contribution will arrive at the end of the editorial assessment process.

**5. Replication**
Since a part of the amount paid for the submission will get locked and allocated in the

SC in case of invalidation, other authors will have the incentive to replicate existing studies. Researchers that contradict the study will get compensated with EKA tokens. This ensures the reproducibility of findings and minimizes wasteful time and financial spending.

**6. Data Analytics**
With the help of well-known text-mining techniques, Eureka will extrapolate the main topic of an article and thus automatically assign a set of reviewers which are experts in that field. In doing so, reviewers with different backgrounds and coming from different fields will have the incentive to join the Eureka community as reviewers.

Eureka tokens are used for:

- Submitting articles for review
- Proving ownership of observations by timestamping them
- Upvoting and downvoting articles with EKA micropayments
- Voting for awards and prizes
- Paying the submission fee, which includes a fee for operating costs and a variable fee to reward two or more peer reviewers
- Automatically connecting peer reviewers with manuscript submissions transferring
- Reward ratings from peer reviewers internally and externally
- Rewarding peer reviewers, authors and funding agencies for citations and for Having exceptional reward ratings
- Research funding using smart contracts: automatically release funds for studies,
- Peer review once conditions have been met
- Token pledges, awards, and prizes contributed by organizations for scientists and researchers to reach set milestones and help solve society's problems
- Crowdfunding from the general public to directly pay scientists around the world to conduct research studies on the Eureka platform into under-researched diseases and ailments

# Chapter 4

# Implementation

In order to ensure a high-quality application over a short period of time, we distributed the responsibilities as follows:

- Smart Contract: Severin Wullschleger
- Convenience Layer (Back-end): Andreas Schaufelbuehl
- Front-end: Lucas Pelloni
- Testing: Andreas Schaufelbuehl

For us important to emphasize is that the upper responsibilities do not mean that the person in charge had developed this part on his own. Moreover it means that that person has the lead and was in charge of creating structure and model of the different responsibilities. The on-hand development of all these different areas is moreover a work of all three members.

We try to describe the platform's implementation from top to bottom, which means we start relatively high level by describing the overall publishing and peer-review process with the help of a business process model. After that we present a short architecture overview before we jump into the different architecture components and the interfaces between them. We conclude with another business process model which illustrates the interaction between all architecture components in a specific use case:

- EUREKA's peer-review and publishing process
- Platform architecture overview
- Front-End
- Smart contracts
- Back-End

- Interface between front-end and smart contracts
- Interface between smart contracts and back-end
- Interaction between all components

## 4.1 The EUREKA peer-review and publishing process

In this section the EUREKA overall process is described with the business process model notation (BPMN).

Figures 4.1 and 4.2 shows the interaction between the main stakeholders of the platform, the smart contracts and the EUREKA convenience layer. It only shows the main process of submitting an article, peer-reviewing the article and deciding if the article is accepted or not. It therefore only shows the stakeholders which perform actions in this process.

The process starts with the author who wants to submit an article. When he submitted the article, a handling editor can assign himself for handling this submission process. It means that he is responsible for the end-to-end peer-review process. He has to perform a sanity check and can, but does not have to, invite expert reviewers. EUREKA distinguishes between expert reviewers and community reviewers. Expert reviewers are approved from the journal as experts in a specific field whereas community reviewers are unknown or doesn't reach the requirements for being an expert. Usually expert reviews are in greater detail and more specific and community reviews are only single annotations on a paper. Technically, the community review can be in great detail as well. Since only a defined number of reviews are rewarded, we want to assure that the reviewers are reviewing the papers carefully and do not rush with their review work just because they want to claim the reward theirs. Therefore expert reviewers can sign up for reviewing a paper, which will result in reserving the reward for a specified amount of time (e.g. three days).

As soon as a review is handed in, the handling editor can read them and accept or decline it. If it is declined, the reviewer has a chance to correct its review. When enough reviews are accepted (required amount can be specified with journal parameters) the handling editor can accept or decline the article. When the handling editor is not happy with the article, he can request a new article version with some corrections. If the author accepts the request and uploads a new article version, then the process starts over again.

The rewards for all stakeholders are just transferred when the submission process is closed.

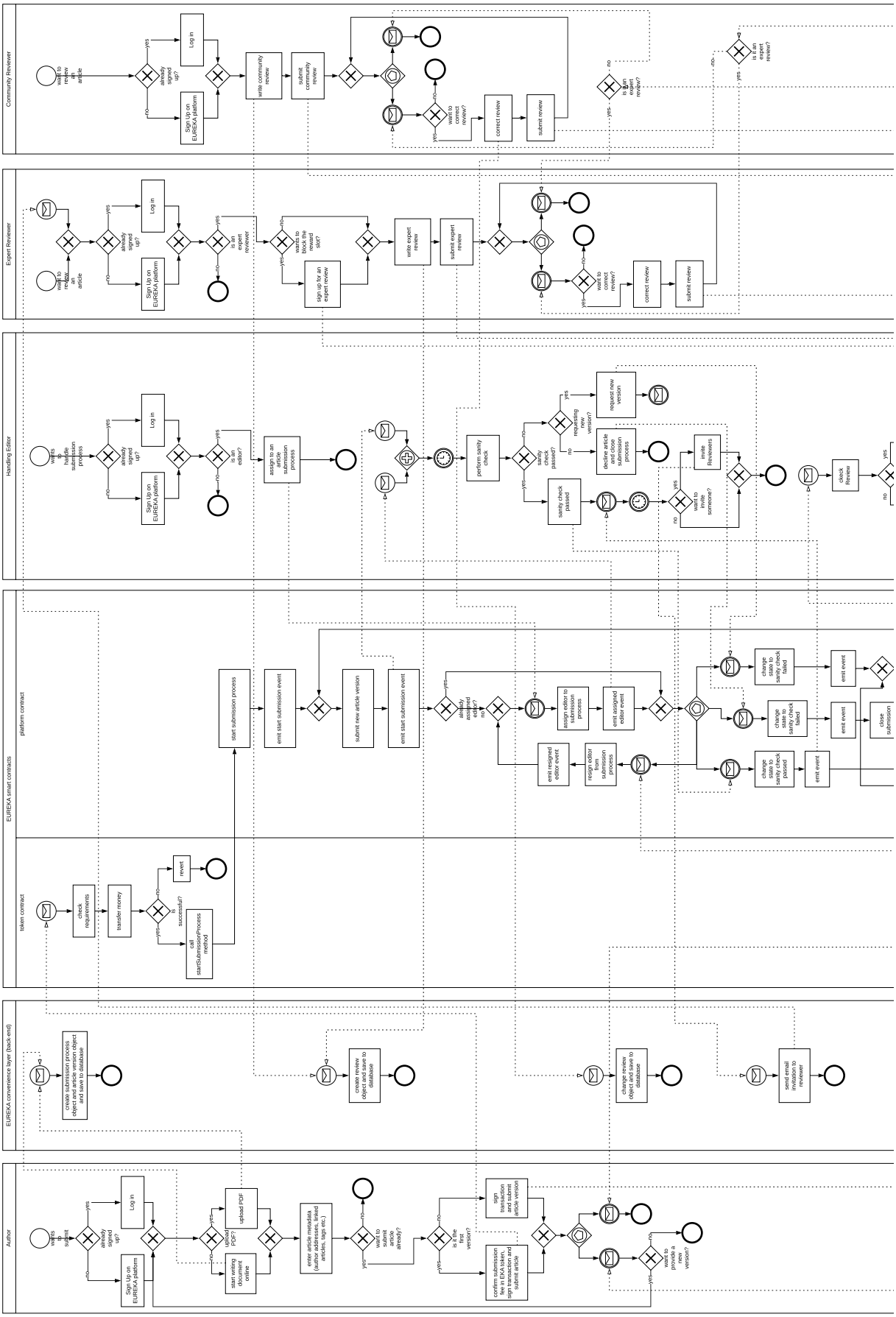



Figure 4.1: EUREKA Platform Process Model, platform-stakeholder interaction, part 1

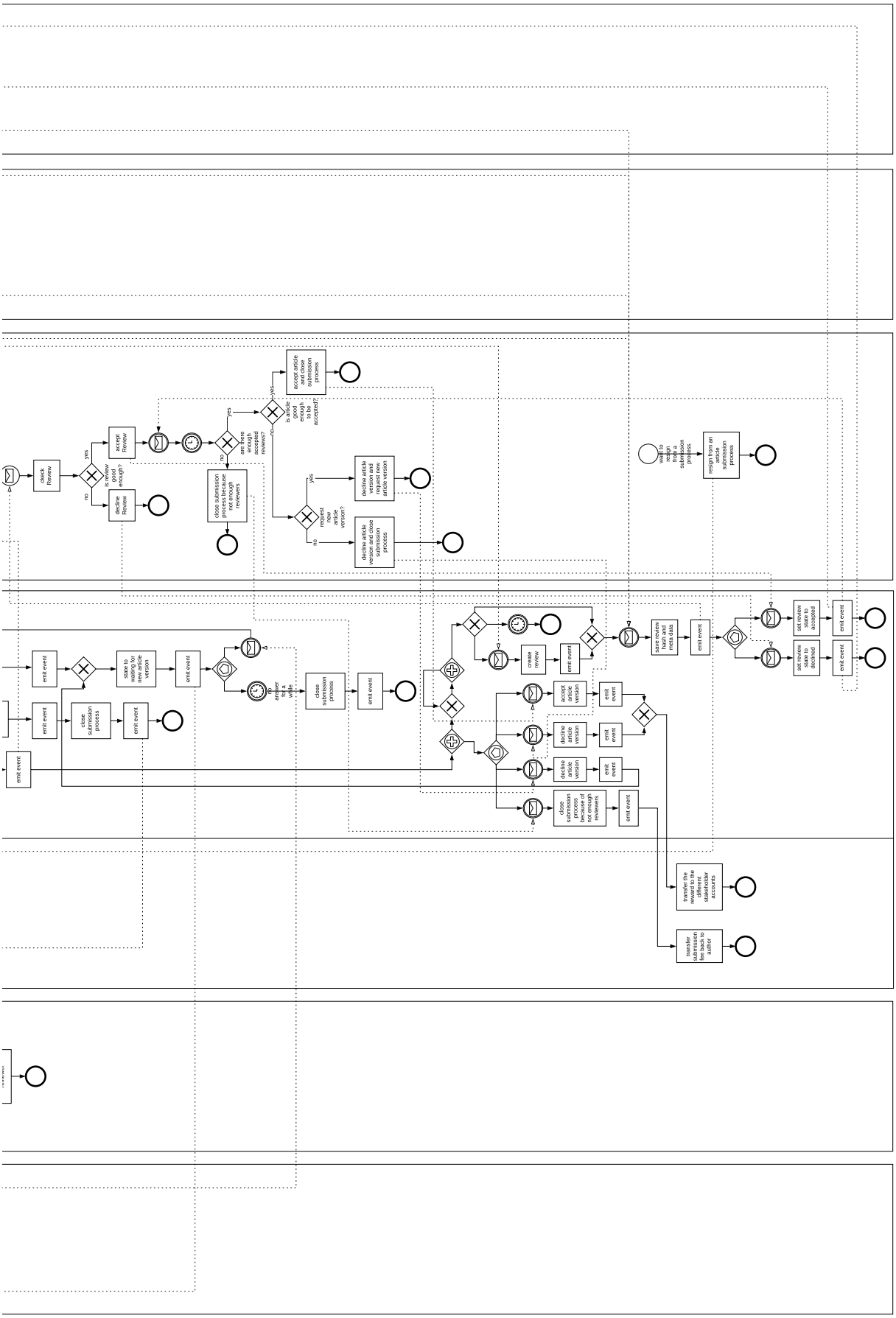



Figure 4.2: EUREKA Platform Process Model, platform-stakeholder interaction, part 2

## 4.2 Platform architecture overview

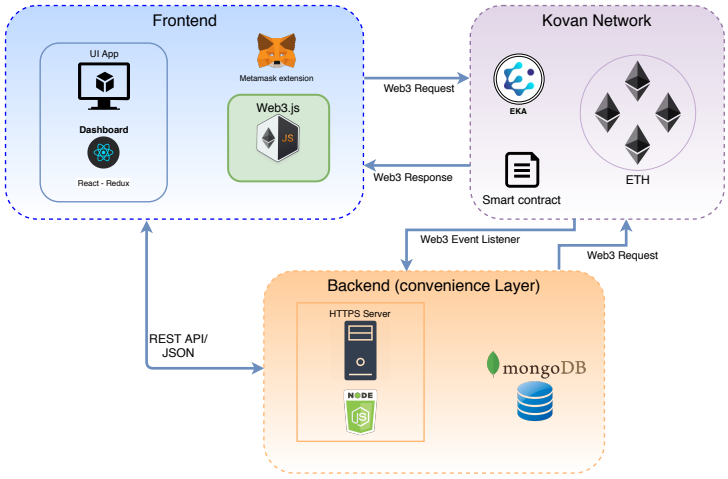



Figure 4.3: EUREKA's Architecture

The platform consist of three main parts:

- **Front-end:** UI and interaction with the user of the platform. It provides access to our platform, where user can register and upload articles, but also sign up and work as an reviewer or editor of articles from other users. Javascript is used together with the library ReactJs [3] for the UX. The front-end application communicates through a REST-API [77] with the back-end application. Further it uses the chrome extension Metamask [15] and the library Web3.js [6] to interact with the Smart Contract (SC).
- **Smart Contract:** Responsible for the token flow. It handles all interaction involving Eureka Token (EKA). It regulates the overall token flow process, meaning that the payment of reviewers and editors of articles is done without any human interaction. Using the Ethereum network [10] as base, the token flow process is regulated through the usage of the in-house token currency EKA.
- **Back-end (Convenience Layer):** Monitors the SC and provides a middle layer for intuitive interaction with the SC. It copies the state of the SC and informs user accordingly. Based on time frame conditions it interacts with the SC, reverting states, where user are not respond. It uses mongoDB [29] to mirror and store the state of the SC. The application runs with Node.js [27] and it also uses the Web3.js [6] library to communicate with the SC.

This chapter provides a in-depth explanation of all the three main parts of our platform. It shows all the functionalities implemented not only on a theoretical and logical level, but also showing implemented code models an example, with the goal to serve the reader

with depth knowledge of the underlying code structure.
Presenting the three main components first, the chapter proceeds with showing the interface and communication between these front-end and SC, followed by the interface of SC and back-end. We consider the understanding of the interaction of these components as crucial for the understanding of the overall process of the platform.

## 4.3 Front-End

### 4.3.1 General information and front-end concepts

For our User Interface (UI), we are using the ReactJs Framework [3]. ReactJs is a JavaScript lightweight library for building fast and efficiently web apps. EUREKA uses React in combination with JSX [59], a XML-like syntax extension built on top of JavaScript which makes the code more readable, faster and easier to maintain. In fact, JSX adds XML elements to JavaScript such as tags, which can have attributes, children and other properties. Each JSX element must be in line with the strictly hierarchical tags structure. Finally, JSX can be easily combined with CSS and HTML [5] and thus add power to the React rendering process.

#### UI

EUREKA has its own design Palette which has been implemented from scratch, including design components (such as Modals, Buttons, Input fields, Tables, Label, etc.) and color schemas. However, concerning the styling of basic design elements, EUREKA's design is inspired by the Argon Design System from Creative Tim [60]. It is worth to point out that this design system is not available for React yet and thus we just integrated some CSS attributes by inspecting its Web page. Indeed, EUREKA does not use any external design Framework but relies on reusable React components. Concerning style maintenance and HTML tags management, EUREKA relies on the well-known external library *styled-components* [43]. This library, enhances CSS in order to style React component systems and provides several advantages [43], such as:

- No more bugs with duplicated Class Names
  The library generates unique class names during the rendering process and ensures that there are no duplicates for different HTML tags.
- Conditional Styling
  CSS elements are like React components. Using props [4] in combination with Template Literals syntax [62] CSS attributes can be dynamically styled without having to manually manage tons of external CSS classes.
- Maintenance
  Searching for styled elements is the same as searching for React components. Everything is compact and stays in the same file. In fact, styled-components avoids time-consuming searches for finding the styling affecting a specific component.

Our design is well-structured, contains where possible variables and place-holders and supports responsiveness for Mobile devices. For general HTML body styling (*e.g.,* the font family shared among all components), a *index.css* file is provided with the corresponding CSS at the top of the application tree. In the next Release, we plan to migrate everything to *styled-components* since they just released a new version that supports dynamically updateable global styles. This will allow to dynamically manage global styles which will ease future white-label processes (*e.g.,* simplify the creation of a new theme for third-parties). At the moment, we still need to mirror some CSS attributes within the *index.css* file into certain reusable React components coupled with *styled-components*. For instance, EUREKA's colors palette is declared both in the *index.css* and *colors.js* files. In the first one, EUREKA' colors are declared as CSS variables in the *root* element. In the second one, they are exported as static JavaScript variables. The Figure 4.4 illustrates the differences between these two files.

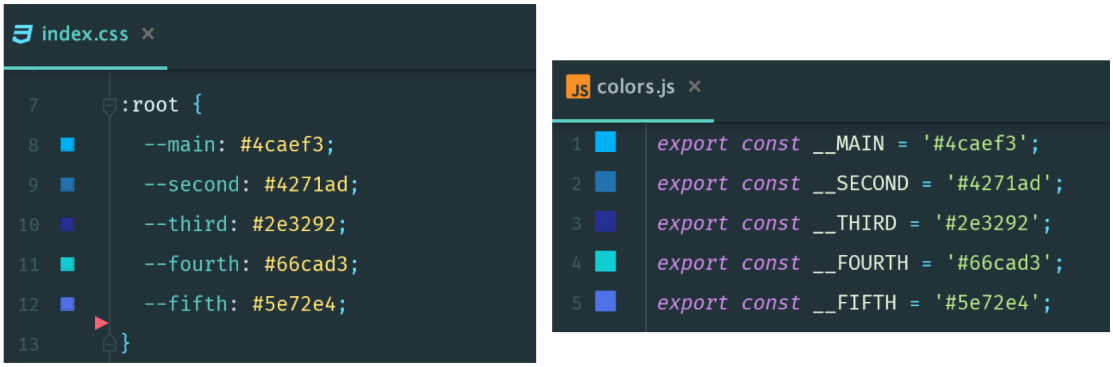



Figure 4.4: EUREKA's Colors.js vs. Index.css

The underscores in the variable names are used for easing the research of the color names within the project.

### Redux and Blockchain Integration

In order to easily manage different states simultaneously, we integrated Redux [34], the most popular state management tool for React and Javascripts applications. Redux helps the application to know what and where is the current application state. Since EUREKA's application data is dynamically shared among different components may be tedious to track every internal state change. Most of the components in the React ecosystem are connected with the Redux Store [46]. Redux wraps a component by injecting the requested states and actions as props. We decided to deeply couple Redux with EUREKA's Front-end for the following reasons [47]:

1. Easily management of predictable and different states
   One of the key advantages of using Redux, it that the state is always predictable. When a Reducer (*i.e.,* a pure function) takes the same action and state, Redux

guarantees that the same result is always produced. Since the state is immutable, it can be changed if and only if an action gets triggered.

2. Debuggable Code
   Using well-known Redux Dev Tools [48], the state can be moved back and forwards and see how a state change affects the application behaviour in real-time. This makes easier to find potential bugs and debug the application.

3. Maintainability
   Redux provides strict guidelines for its integration by having a specific structure and syntax. However, for someone who already has experience with the Redux ecosystem, makes the code easier to understand and thus to maintain.

### Structure

EUREKA's Front-end [8] is dividend among more than 120 React components, where a component can be either functional, *i.e.* a stateless view [42] that displays a set of passed properties (props) [4] or a class, *i.e.,* a stateful component [42] that encapsulates and manages its own state [4] and logic. Usually, the latter are those components which are in charge of sending asynchronous requests to our Back-end, change the state that is locally encapsulated within the component and pass the fetched data through the application tree using read-only props.

In other words, stateless components are responsible for *"How things look"* in your application, while stateful components are responsible for *"How things work"* by considering dynamic states and actions that can change at any time in the components life cycles. In addition to React components, there are different JavaScript modules that interact with our Front-end. Web3 [6] components are in charge of the interaction with the SC. They contain specific functionalities for the Etheruem ecosystem and allow you to interact with a local or remote Ethereum node. Finally, Helpers function makes the code more readable by giving descriptive names to methods that can be reused in the whole project. To summarize, the file structure that has been adopted for our project can be divided in three main categories:

- **React components**
  React components are used to split the UI into independent and isolated pieces that can be reused along the application. Conceptually, React components are like native JavaScript functions that take props [4] as input and renders React elements into the screen. This category mainly contains stateless and stateful React components and can be divided into six sub-categories:

  – *Stateful components, React Routers* [41]
    Stateful components represents the logical layer for EUREKA. They are responsible for storing application data and defining the different behaviour of the child components based on an internal state change. They can be presentational (*e.g.,* manage the state for an input field) or logical (*i.e.,* fetching the data from our server and pass these data along the application tree).

In addition to the classic stateful components, this category contains React Routers as well. React Routers are built at the top of React and are used for core dynamic routing in our Front-end. They defined, based on certain conditions such as matching a URL path, which component will be rendered.

– *Constants*
This category consists of a variety of Javascript exported variables which can be statically shared along the platform. For instance, a string for a common message in a Modal or the EUREKA colors palette.

– *Design components*
Styled components [43] represents the foundation of EUREKA's design. This module enhances normal CSS and removes the mapping between components and styles by creating normal React components instead of external CSS classes. EUREKA exploits this module in order to have reusable styled components that can be shared along the whole platform and thus improving the style consistency. For instance, a Button or a Input field can be imported into different React components and have the same style structure. Based on different props, they can be conditionally styled as well. The following code snippets illustrates an simplified example of how React components coupled with styled-components can be conditionally styled.

```
1  const ParentContainer = styled.div`
2    // other CSS attributes here
3    display: ${props => (props.isHidden ? 'none' : 'flex')};
4  `;
5
6  const TextContainer = styled.div`
7    // other CSS attributes here
8  `;
9
10 export default class Notification extends Component {
11   constructor() {
12     super();
13     this.state = {
14       isHidden: false
15     };
16   }
17   hideAlert() {
18     this.setState({isHidden: true});
19   }
20   render() {
21     return (
22       <div>
23         <ParentContainer isHidden={this.state.isHidden}>
24           {/*Other HTML tags..*/}
25           <TextContainer onClick={() => this.hideAlert()}>{
                 this.props.text}</TextContainer>
26           {/*Other HTML tags..*/}
27         </ParentContainer>
28       </div>
29     );
30   }
31 }
```

The code snippet above shows a stateful React Component that stands for a simple Notification label in our Front-End. As you can see, there is no mapping between components and styles using external CSS classes, but everything can be represented in the same component. Indeed, in the first part (until line 12) there are only native HTML tags overwritten with a styled-components instance called *styled* (line 1 and 9). Starting from line 13, we can see the logic and rendering process of the stateful component. The conditional statement in order to have different styling behaviours can be found at the lines 3-5. Using Template literals syntax [62] a condition can be embedded in the expression for the specific CSS attribute. For instance, the *ParentContainer* passes as props to its styled-container the internal state that describes whether a certain notification in the Front-End has been hidden or not. As specified in the constructor, the initial state is set to false which means that the notification is displayed at the beginning. Based on the result returned by the binary operator [63] at line 3, the CSS expression that gets rendered for the *ParentContainer* is the following:

```
1 display: flex;
```

As soon as the user clicks on the *TextContainer* the internal state changes (line 17-19) and thus the whole component gets re-rendered. This leads to get the first operand as a result for the binary expression for the *display* attribute (and so hiding the notification in the UI):

```
1 display: none;
```

As you can see, conditional styling with styled-components can be done using Template literals syntax [62] and binary operators [63] and so allow to save a substantial amount of work that external CSS classes would require.

– *Reducers*
This section is composed by Redux [34] components: Reducers [45], Store [46] and Actions [44]. Redux is a well-known predictable state container for JavaScript-based applications that allows your application to easily manage different local and global states among different components. Actions are used for payloads of information that are directly sent to the global store, which in turn holds the different application states together. Reducers define how a state shall change when a certain action is executed. In other words, an action can be viewed as a trigger for a reducer which defines how an application's state changes. The store keeps track of the application states and gives access to different component to access them in the application tree. An example is shown in the code snippet below. The code illustrates a simplified version of the *Account* component, that is in charge of displaying the account information of the user which is logged into the platform. Since these type of information are often consumed by different components, they have been stored inside the Redux store in order to be easily shared. The connection to the Redux store happens between line 27 and 32. As you can see, the Redux store injects into *MyAccount* two additional props, which are (i) the account that has been unlocked by the User with MetaMask and therefore exposed to the Web3.js instance and (ii) the user object which contains all user-related informations (such as the

email address, the avatar, the name, etc.). The component is therefore able to access these source of information by invoking *this.props* within the whole component.

```
1
2  class MyAccount extends Component {
3
4    // More methods here..
5
6    render() {
7      return (
8            <Card>
9              <CardHeader>
10               <Avatar> src={this.props.user.avatar}</Avatar>
11               <Email>{this.props.user.email}</Email>
12             </CardHeader>
13             <CardBody>
14               <Balance>
15                 {this.props.metamaskAccount.EKABalance} EKA
16               </Balance>
17               <Balance>
18                 {this.props.metamaskAccount.ETHBalance} ETH
19               </Balance>
20             </CardBody>
21           </Card>
22     );
23   }
24 }
25
26 // Connection to Redux Store
27 export default connect(
28   state => ({
29       user: state.userData.data,
30       metamaskAccount: state.accountsData.selectedAccount
31   })
32 )(MyAccount)
```

It is worth to point out, that usually stateful component do subscribe to the Redux state and dispatch different actions through the application while presentational views read data and invoke callbacks using props.

– *Stateless components (i.e., views) and Guards*
This category represents the foundation of EUREKA's front-end. In fact, there are more than 50 views in our application. Conceptually, stateless components do not manage any internal state and do not have the component lifecycle methods [42]. They just take props as an argument and returns a react element. In EUREKA, they are used as presentational views for components like *Address.js*, *Avatar.js*, *Figure.js*, etc. In fact, the main purpose of these components is how to display the input data in the UI and not how to manipulate them. They can be viewed as a sort of black-box components, where an input comes in and the output results into a styled view. For instance, the *Address.js* view can be used to display the address of the logged User in the Header as well as the sender address of a Ethereum transaction.

```
1
```

```
const SpanContainer = styled.span`
  color: ${props => props.color};
  font-size: ${props => props.fontSize};
  word-break: break-all;
`;

const Address = ({ethereumAddress, content, ...otherProps}) => {
  return (
    <SpanContainer href={`/app/users/${ethereumAddress}`} {...otherProps}>
      {content}
    </SpanContainer>
  );
};

export default Address;
```

As you can see, the *Address* view can be completely reused and anonymised since it just takes a set of parameters and returns a React element displaying the passed content that points to an external link.

In addition to traditional Views, this category includes Guards as well. Guard is a *angular-like* term. They are stateless components that protect certain routes within the application that can be accessed if a set of needed criteria are fulfilled. For example, the *SC Guard*, is a stateless JavaScript function which ensures that the SC route remains protected and can be accessed if and only if the involved user is the SC owner.

```
// Guards.js
export const ContractOwnerGuard = ({user, ...otherProps}) => {
  if (user.roles.includes(Roles.CONTRACT_OWNER)) {
    return otherProps.children;
  } else {
    return (
      <Redirect to={{pathname: '/app', state: {from: otherProps.location}}} />
    );
  }
};

// DashboardRouter.js
class DashboardRouter extends Component {
  render() {
    return (
      <div>
        {/*Other Routes here..*/}
        <Route
          path={`${this.props.base}/owner`}
          render={() => (
            <ContractOwnerGuard>
              <ContractRouter (...) />
            </ContractOwnerGuard>
          )}
        />
      </div>
    );
```

```
28   }
29 }
30 export default DashboardRouter;
```

The code snippet above illustrates the SC guard coupled together with the Route in the *DashboardRouter.js*. The latter is in charge of rendering the SC Router that in turn renders its sub-routes. As you can see the guard is a JavaScript function that uses the destructuring assignment [64] for unpacking properties, that have passed as props to this function, into distinct variables. The Guard starts with a condition checking which roles have been assigned to the injected user (line 3). If the roles set of the user includes the enumeration variable *CONTRACT_OWNER*, the children components of the *ContractOwnerGuard* are rendered (line 4). In this case, the *ContractRouter* would be rendered. In other words, the SC routes can be accessed if and only if the user is the SC owner. Otherwise, another user without that specifc role who accidentally tries to access the *owner* route, would be redirected to the main app (lines 7-8).

- **Web3 Modules**
  The Web3 [6] modules represent the core integration layer with the EUREKA SCs. This category is in charge of (i) setting up the needed web3 Instance in order to have a proper configuration for the SC, (ii) providing utility functions for rendering the right context (*e.g.,* identify the current Ethereum network, fetch the unlocked accounts and the corresponding Ether balances) and (iii) exposing a collection of web3 methods for EUREKA's SC which can be imported into other components. This category is highly coupled with the SC folder and its corresponding exported methods. The Web3 initialization process is discussed in the next sections.

- **Helpers Modules**
  This section consists of a variety of utility elements, *i.e.,* descriptive methods or variables that can be reused among the whole application. The majority of these elements, are highly related to the style and layout of EUREKA. For instance, this category contains the *colors.js* and *layout.js* file. The first one, describes the entire color palette by exporting the hexadecimal representation of the EUREKA colors. The second one, contains a collection of CSS Media queries [50] in order to have different screen breakpoints. By doing so, EUREKA's design components behave differently on each side of the breakpoint and make the EUREKA's Front-end compliant with the Responsive Web Design principles [50].

The following Application tree illustrates a simplified version of the EUREKA's Front-End structure.

```
└── frontend
    ├── helpers
    ├── web3
    └── webpack
        ├── components
        └── constants
```

```
    design-components
    reducers
    views
  AppJs
  ReduxInitJs
```

*App.js* and *ReduxInit.js* are the root components of the React application respectively of the Redux Store. Indeed, the React DOM renders at the Top Level of the React API and has one child which is the *ReduxInit.js*. This component is in charge of initializing the Redux Store [46]. In its turn, this Redux component renders the *App.js*, which is the root component of the whole React application.

### Interconnections

The Figure 4.5 illustrates the high-level inter-dependencies within the Front-End ecosystem. React components represent the foundation of EUREKA's Front-end and can be stateless as well as stateful [42]. As explained in the sections above, the Redux [34] ecosystem consists of three main elements which are *Actions*, the *Store* and *Reducers*. User interaction is the first trigger for actions coupled with the Redux ecosystem (*i,e.,* clicking a button for loading the user profile). Actions in Redux can be viewed as payloads of information [44] for the Redux store [46]. They always need a literal type that indicates which action has been informed. By doing so, the store is aware of what type of action has been dispatched. The Redux store, which holds the global application state, informs the corresponding reducer about the new action by passing as input data the action's type together with the previous application state. The reducer sends back to the store the new application state in response to the action's type that it has been passed to it and informs the corresponding React component about the new state. In other words, actions describe what happened in the application and reducers how the application's state changes where the store is the bridge between normal React components and the Redux Framework. Redux is the global state manager for EUREKA's Front-End but does not manage the Web3.js [6] Instance. In fact, the Ethereum JavaScript API instance is defined and configured at the top of the React application tree. Its configurations strictly depends on the Ethereum Provider which is detected once the applications gets started. The following code snippet illustrates the initialization process and configuration of Web3.js instance in our Front-End.

```
1  /********************** Init Web3 **********************/
2  import platformABI from '../GanachePlatformContractABI.json';
3  import tokenABI from '../GanacheTokenContractABI.json';
4  import platformAddress from '.  ./GanachePlatformContractAddress.json';
5  import tokenAddress from '../GanacheTokenContractAddress.json';
6     (...)
7     const web3 = window.web3;
8      let web3Instance = null;
9      let platformContract = null;
10     let tokenContract = null;
11     let provider;
12     if (typeof web3 !== 'undefined' && web3.currentProvider.isMetaMask)
          {
```

```
13        // MetaMask as main provider
14        web3Instance = new Web3(web3.currentProvider);
15        platformContract = new web3Instance.eth.Contract(
16          platformABI,
17          PLATFORM_KOVAN_ADDRESS
18        );
19        tokenContract = new web3Instance.eth.Contract(
20          tokenABI,
21          TOKEN_KOVAN_ADDRESS
22        );
23      } else {
24      // Ganache (dev) as main provider
25        web3Instance = new Web3('http://localhost:7545');
26        platformContract = new web3Instance.eth.Contract(platformABI);
27        tokenContract = new web3Instance.eth.Contract(tokenABI);
28        platformContract.options.address = platformAddress;
29        tokenContract.options.address = tokenAddress;
30      }
```

As shown in the code snippet above, there are two possible scenarios that may happen during the initialization process. The first one occurs when (i) the global *window* [9] interface is able to detect a web3 object which has been exposed to JavaScript in a tabbed browser and (ii) this object has an attribute which indicates that *MetaMask* [15] is the current Provider. In other words, it occurs when MetaMask has been installed as a browser extension. MetaMask is a well-known browser add-on that allows you to run Ethereum dApps right in your browser without running a full Ethereum node. In addition, it helps the common user by providing a user interface to manage crypto identities among sites and sign blockchain transactions. Since EUREKA uses Ethereum [10] as main distributed infrastructure, it requires MetaMask for performing transactions within the ecosystem. However, despite MetaMask is the most popular crypto extension, EUREKA does not rely only on its functionalities. Indeed, in the future we are planning to have our own in-app secure identity vault.

The second scenario occurs when no web3 browser add-on is found. In this case, we apply our fall-back strategy. We call it fall-back, since we have to run our own Ethereum node for instantiating the web3 object and thus executing transactions within EUREKA. In a development and *local* environment, we chosen Ganache [16], which is a personal Ethereum blockchain which you can use to run tests, execute commands, and inspect state while controlling how the chain operates. We can control several accounts in the same test network and quickly see the current status of their balances and monitor which transactions they performed. On the other hand, in a development and *live* environment we can instantiate the web3 object with the address of an Ethereum node we control. EUREKA has such node for the INFURAnet [17] and more in specific the Kovan test network, which is a live test network for Ethereum which has as protocol JSON-RPC over HTTPs.

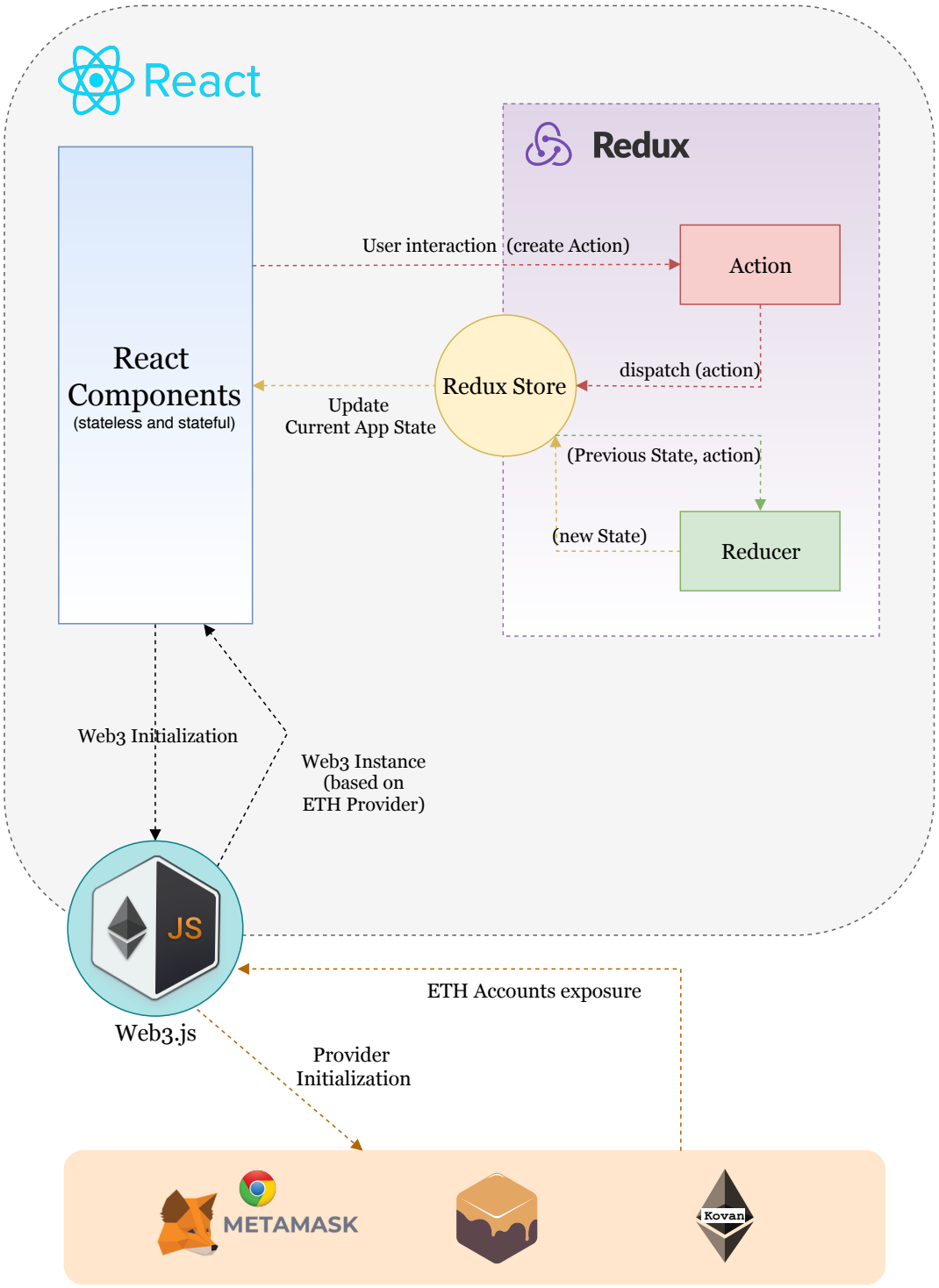



Figure 4.5: EUREKA's Front-End Architecture

### Authentication Process

In the previous section, we mentioned that MetaMask [15] is able to sign blockchain transactions. In addition to that, MetaMask can also sign Ethereum accounts using users private keys. We exploit this feature for the EUREKA's authentication process. Nowadays, modern web apps require an user identifier (*e.g.,* an email address, an username) and a password as input data for their authentication mechanisms. Alternately, the authentication process can be performed through a social media provider. In this sense, EUREKA has a different approach for solving the authentication wall. Indeed, in order to be able to access the dashboard in EUREKA's main app, an user will just need a Ethereum key pair for unlocking its wallet, *i.e.,* a valid ethereum public-private key combination. On a technical level, the web3 instance will trigger MetaMask asking the user for the permission for signing its account using its private key. Furthermore, MetaMask will show an additional and pre-defined message for augmenting the signing process. Afterwards, it will generate a fixed-length string (*i.e.,* a hash) that we will use as user identifier in our database. In that sense we will generate a very strong and secure password for the user, but the private key will remain unknown to EUREKA.

### DraftJS Online Editor

Once the authentication process has been successfully completed, the user comes to main dashboard. At this moment, on an author perspective, a new draft can be created. For that, the user uses our online editor. The editor is built on top of Draft.js [18], where for each section of a document (Abstract, Introduction, etc.) an instance of an *EditorState* [19] is defined. Every time when the draft is sent to the database, the *EditorState* instance is converted into plain JSON and then converted back. In other words, the document gets serialized and then deserialized. At the current state of the platform, the online editor is still a work in progress. Citations and equations cannot be added yet and a minimal amount of scientific fields can be specified. Despite the low customization level of the online editor, authors can add co-authors for the article, specify the title, upload figures, add the relevant disciplines and keywords and have a look the article's preview. In addition to that, draft auto-saving has already been implemented. Once an author click on the submit button, the draft gets *canonically* hashed using SHA256 [20], which is a well-known cryptographic hash algorithm. Since the drafts can be viewed as huge JSON objects [21], canonical JSON is designed to provide repeatable hashes of JSON values. Usually, it is designed for applications that need to hash, sign or authenitcate JSON data structures., including embedded signatures [22]. After that, EUREKA's UI will show the author a set of parameters, which will be used as input data for the smart contract transaction. These parameters are the article hash, the Ethereum addresses of the authors and the url of the article's preview. In the future, referenced authors will be added. Once the author confirm these parameters, an http request is sent to our convenience layer for changing the article's state and storing its hash in our database. Synchronously, our front-end sends a transaction to the EUREKA's smart contract. Afterwards, it shows the transaction hash so that each author is able to track the transaction's status and see if the relevant metadata are stored on the Ethereum blockchain. Since our convenience layer monitors

all events in the smart contract, it recognises that a new submission triggers the smart contract. So it looks for a possible match between the hash saved in the database and the hash stored in the blockchain. In doing so, Eureka's convenience layer is able to keep the front-end updated about the current state of the submitted draft and the executed transactions.

### 4.3.2 Working product (Screenshots)

#### Theme

The following color scale, illustrates the color schema that has been chosen for EUREKA.



Figure 4.6: EUREKA's color schema

Figure 4.7 shows the welcome screen of the EUREKA's platform. An user is able to chose between different options or to press the button "get started".

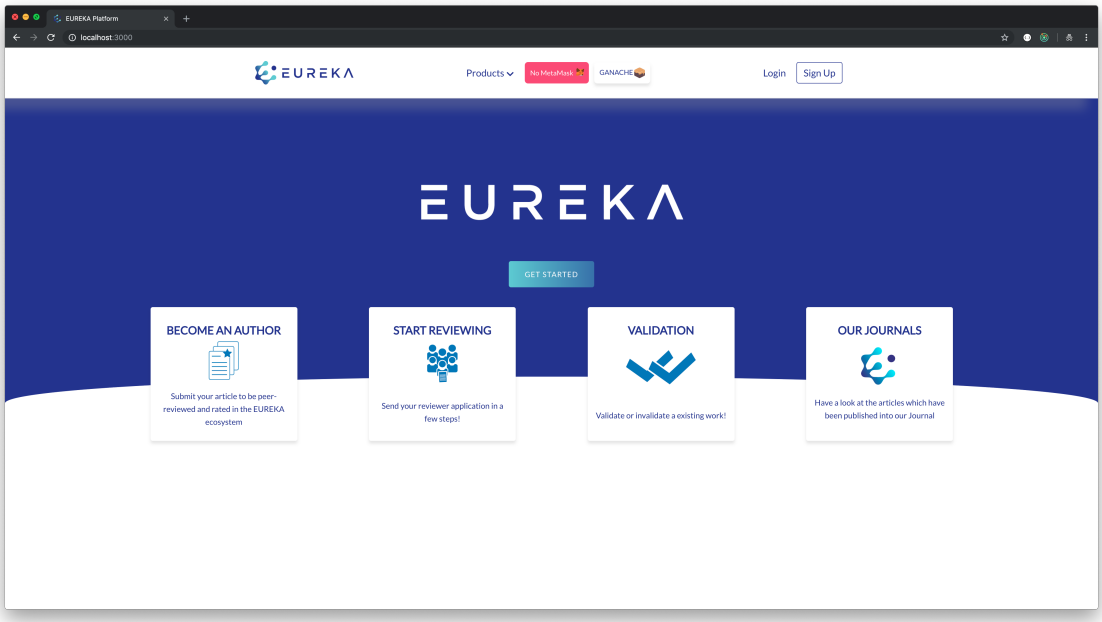



Figure 4.7: Welcome Screen

### Authentication

Figure 4.8 and 4.9 illustrate the screens responsible for the authentication process. The authentication process can happen with MetaMask [15] (in production) or with Ganache [16] (in development). Indeed, in the header there is a label representing the provider that has been chosen and the network that has been detected.

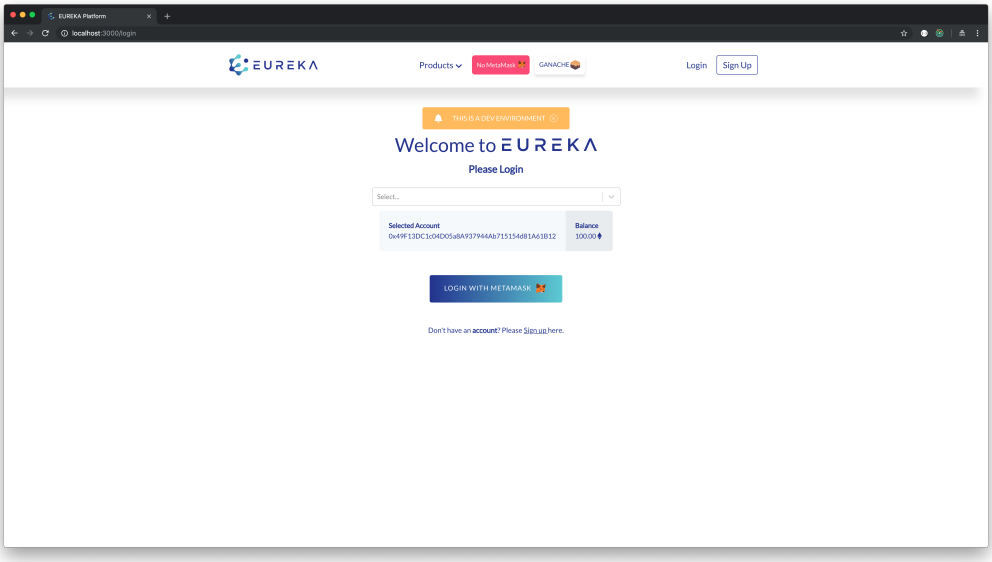



Figure 4.8: Login Screen

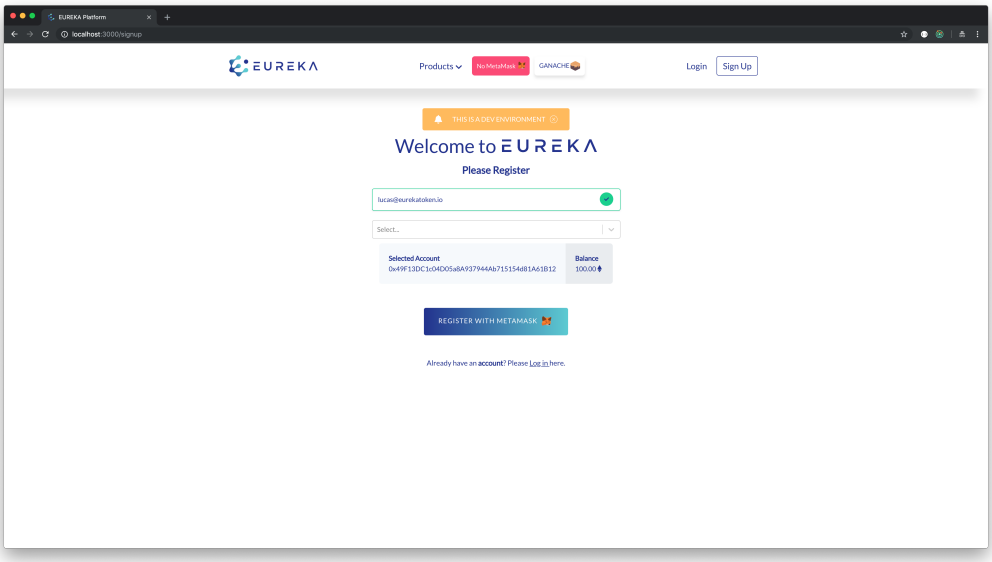



Figure 4.9: Register Screen

### Dashboard

Figure 4.10 illustrates the Dashboard of the platform. In this screen, an overview of the different actions (*e.g.*, the last article that has been submitted) can be found.

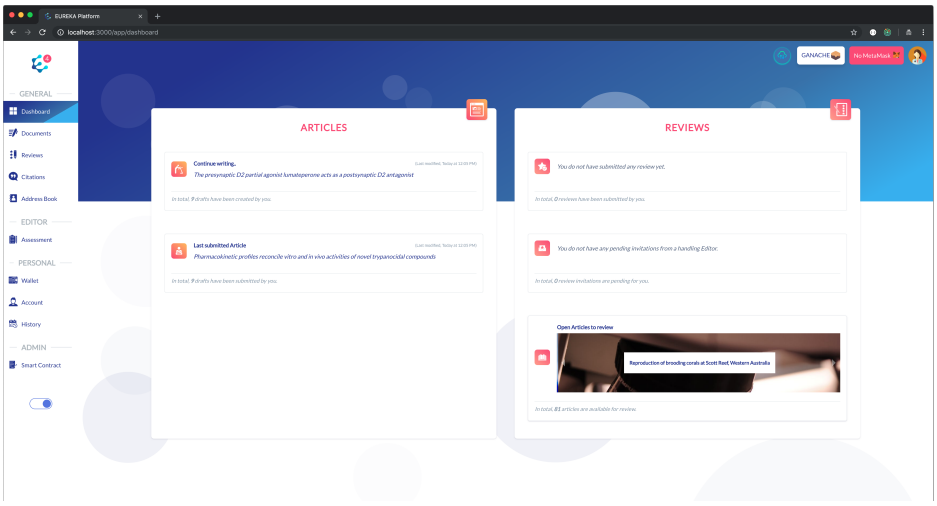



Figure 4.10: EUREKA's Dashboard Screen

### Author Workflow

Figure 4.11 lists the different articles that have been created by the user and have not been submitted yet.

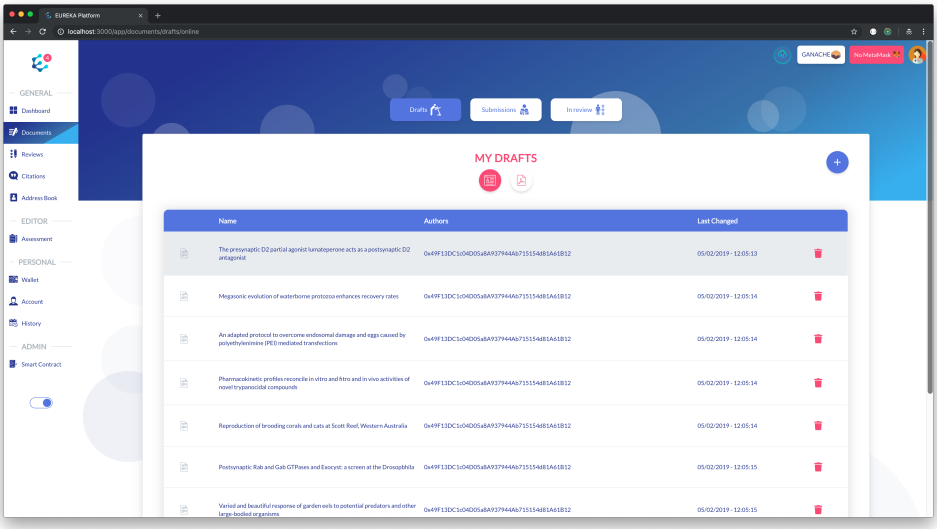



Figure 4.11: Author Drafts

Figure 4.12 lists the different articles that have been successfully submitted by the user.

The ”status” column shows whether the synchronization with the Ethereum Blockchain successfully happened.

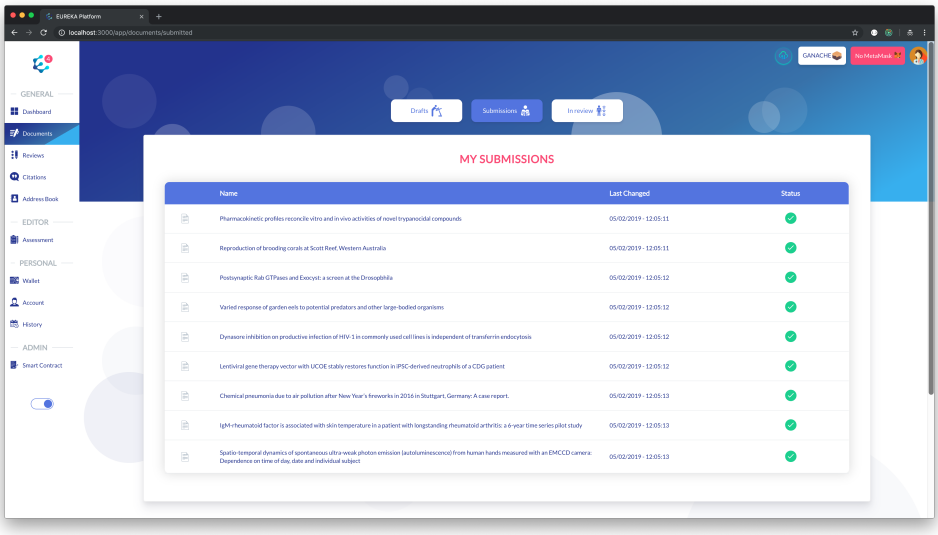



Figure 4.12: Author Submissions

Figure 4.13 and Figure 4.14 illustrate the Online Editor built with Draft.js. With it, authors are able to write their articles directly in the Browser, link other authors and related articles, add pictures, keywords, scientific fields, etc. Once the article is ready, authors are able to submit it. Figure 4.15 shows a Modal listing article-related data, that are used in a second step as input data for the Smart Contract.

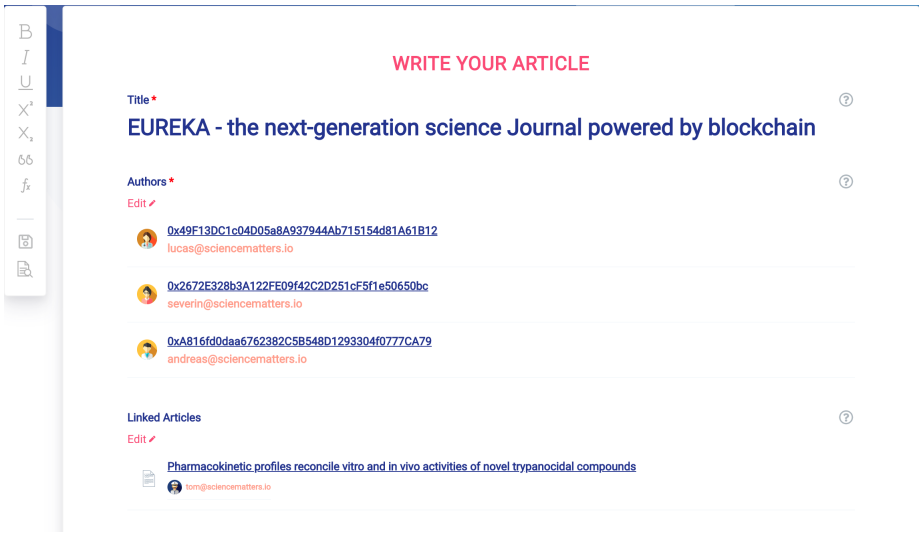



Figure 4.13: Online Editor - Upper Part

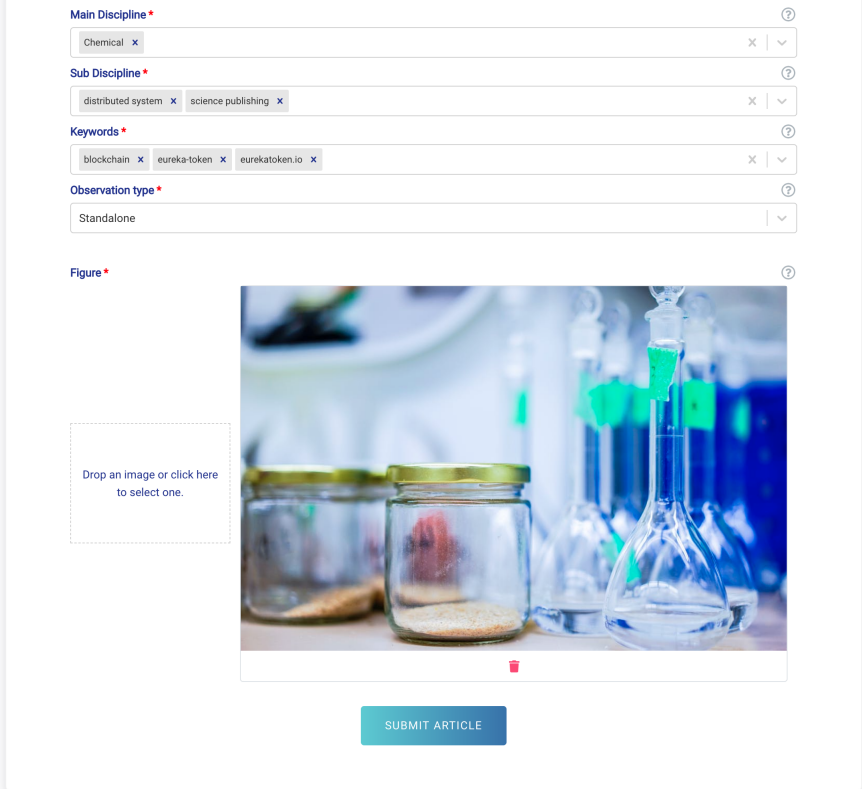



Figure 4.14: Online Editor - Lower Part

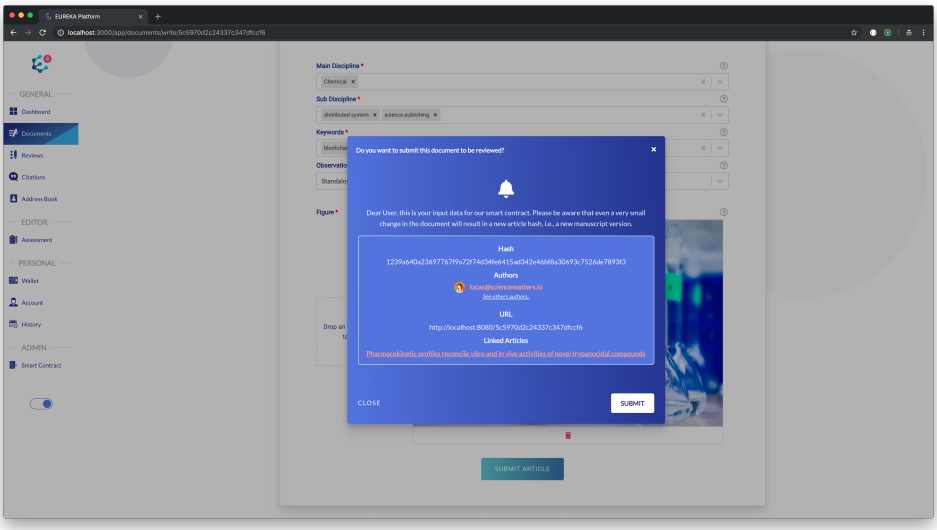



Figure 4.15: Online Editor - Smart Contract Input Data

### Address Book

Figure 4.16 illustrates the Address book of the EUREKA platform. With it, users are able to maintain their own address books which contain a mapping between Ethereum addresses and people's names, affiliations, labels, etc.

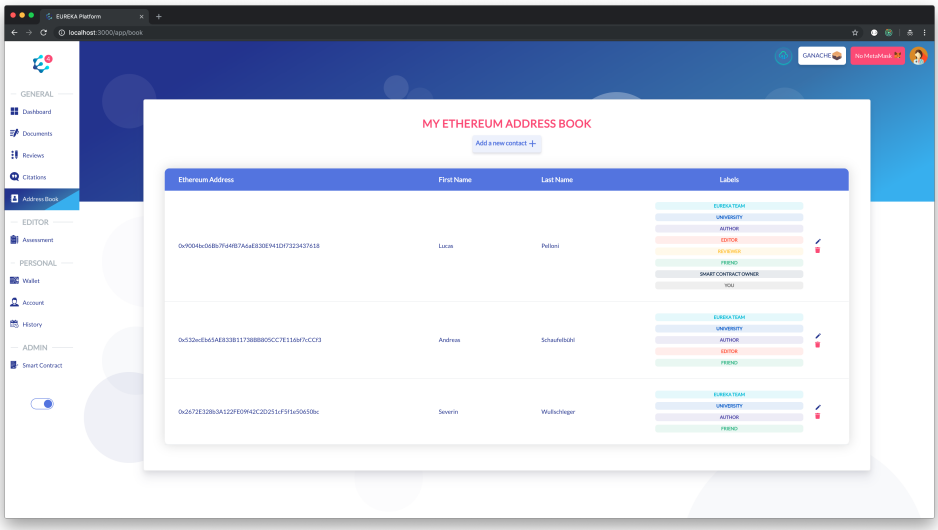



Figure 4.16: EUREKA's Address Book

### Editorial Assessment

*Assignment* Figure 4.17, 4.18, illustrate the first step of the Editorial Assessment process. Editors are able to browse a list of available articles, read them and assign some to him/herself. Everything is publicly recorded on the Blockchain and illustrated with a Transaction Pool (Figure 4.19 and Figure 4.20)

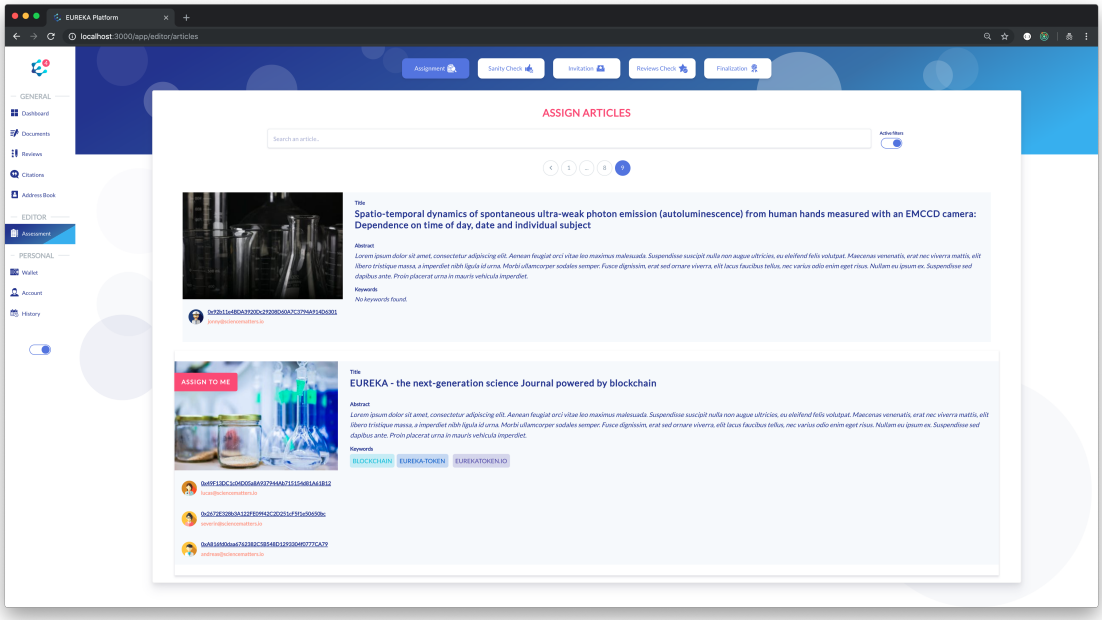



Figure 4.17: Editorial Assessment - Article Assignment

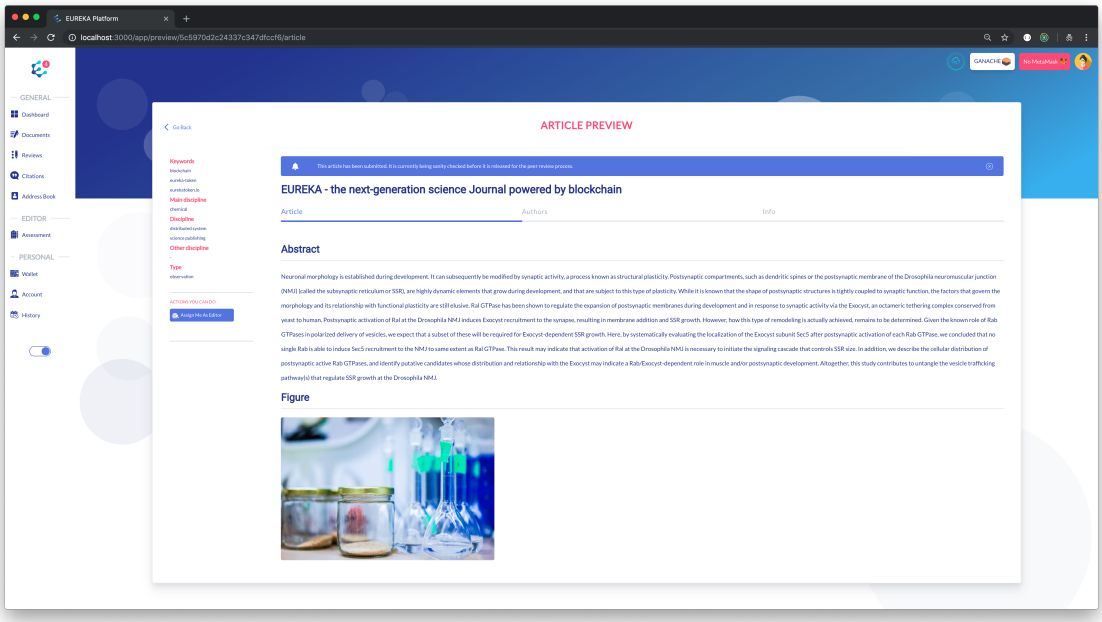



Figure 4.18: Editorial Assessment - Article Assignment - Article Preview

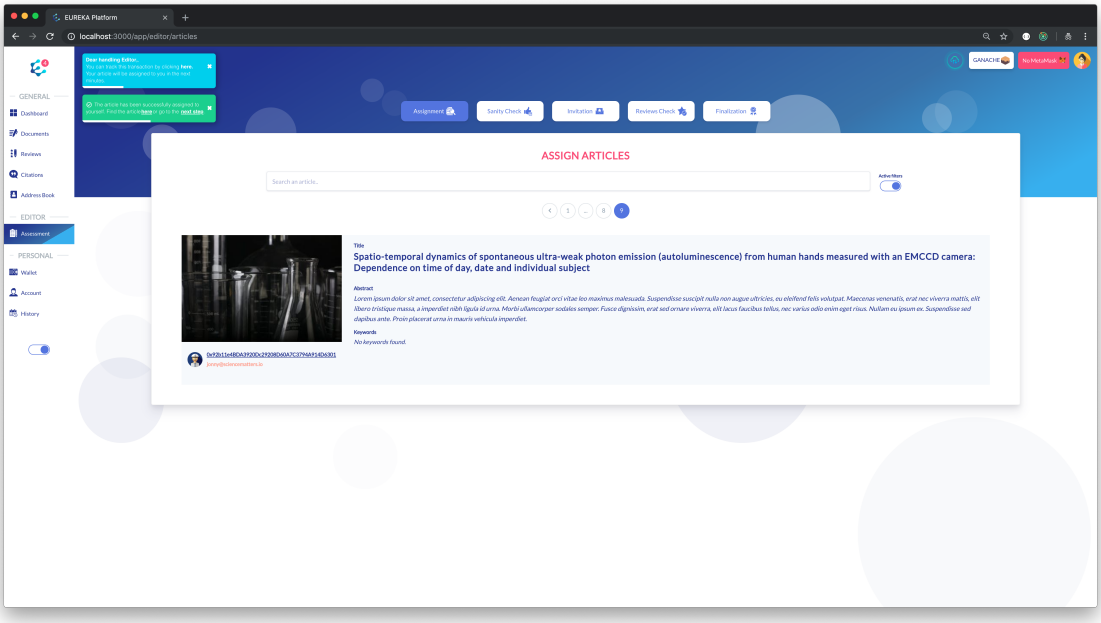



Figure 4.19: Editorial Assessment - Article Assignment - Blockchain Synchronization

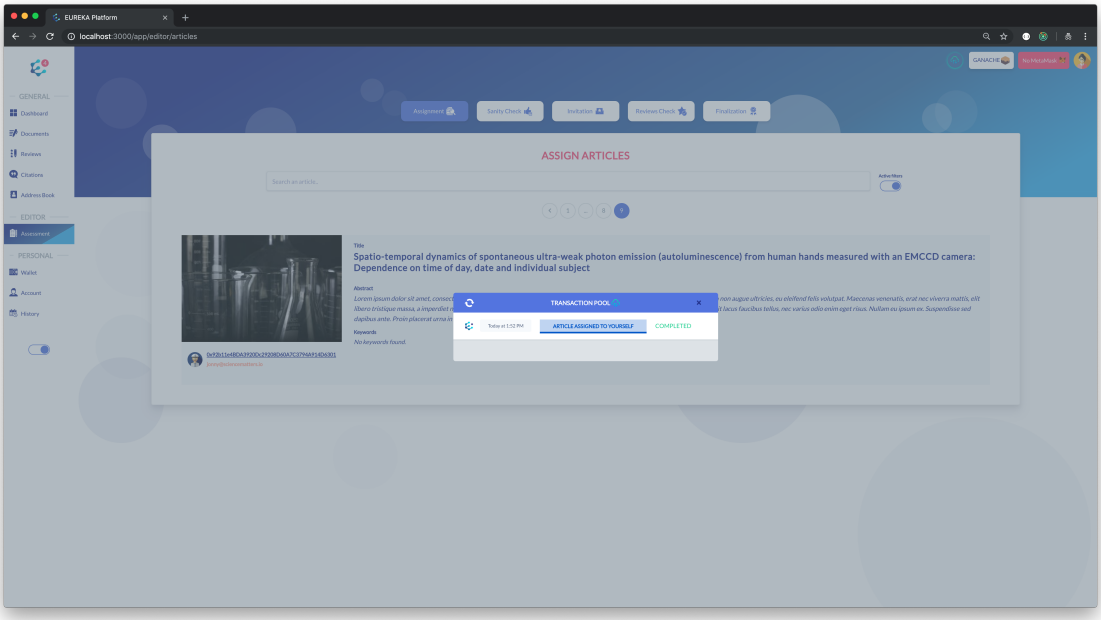



Figure 4.20: Editorial Assessment - Article Assignment - Transaction Pool

*Sanity Check* Figure 4.21 and 4.22, illustrate the second step of the Editorial Assessment process. In this step, editors are able to perform sanity checks, *i.e.*, quickly evaluate whether the articles are ready for the reviewing process.

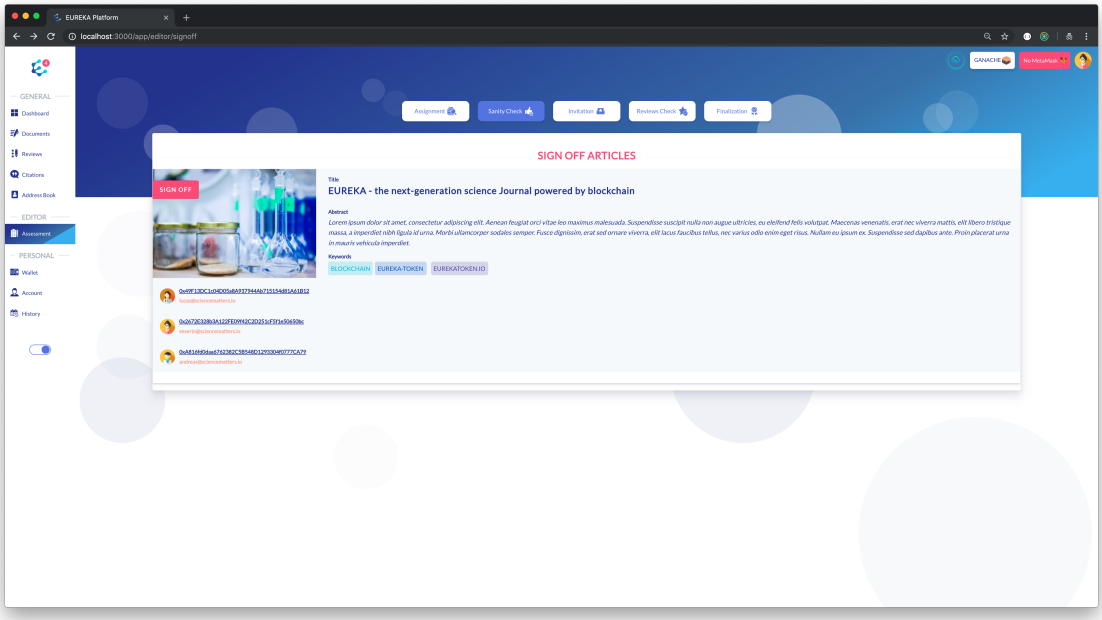



Figure 4.21: Editorial Assessment - Sanity Check

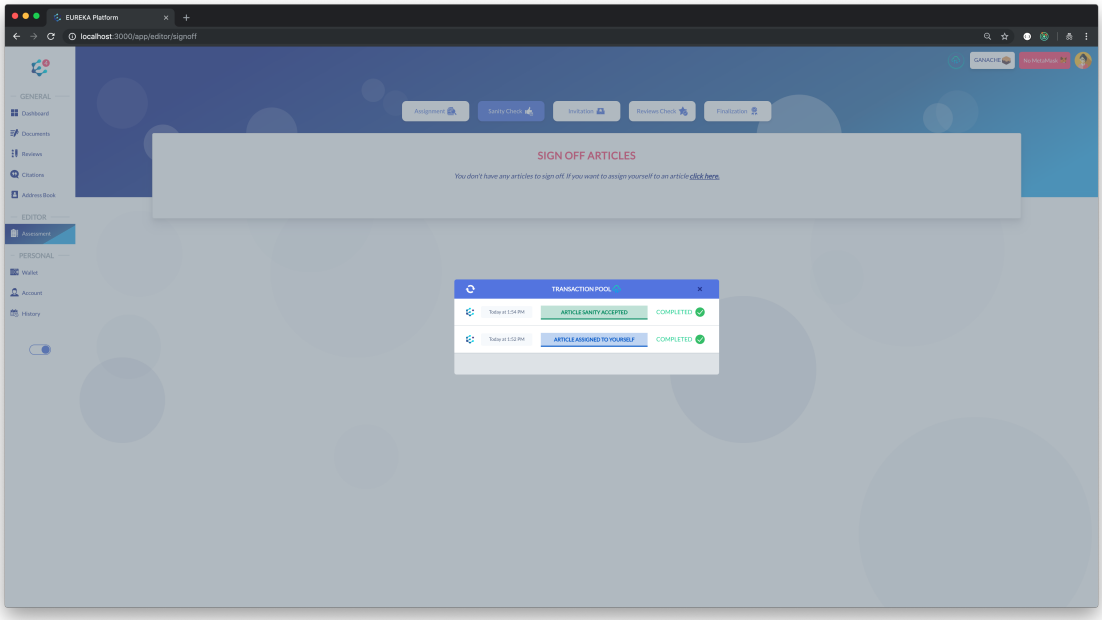



Figure 4.22: Editorial Assessment - Sanity Check - Transaction Pool

*Reviewers Invitation* Figure 4.23 and 4.24, illustrate the third step of the Editorial Assessment process. In this step, editors can invite expert reviewers that they believe are adequate for reviewing the article.

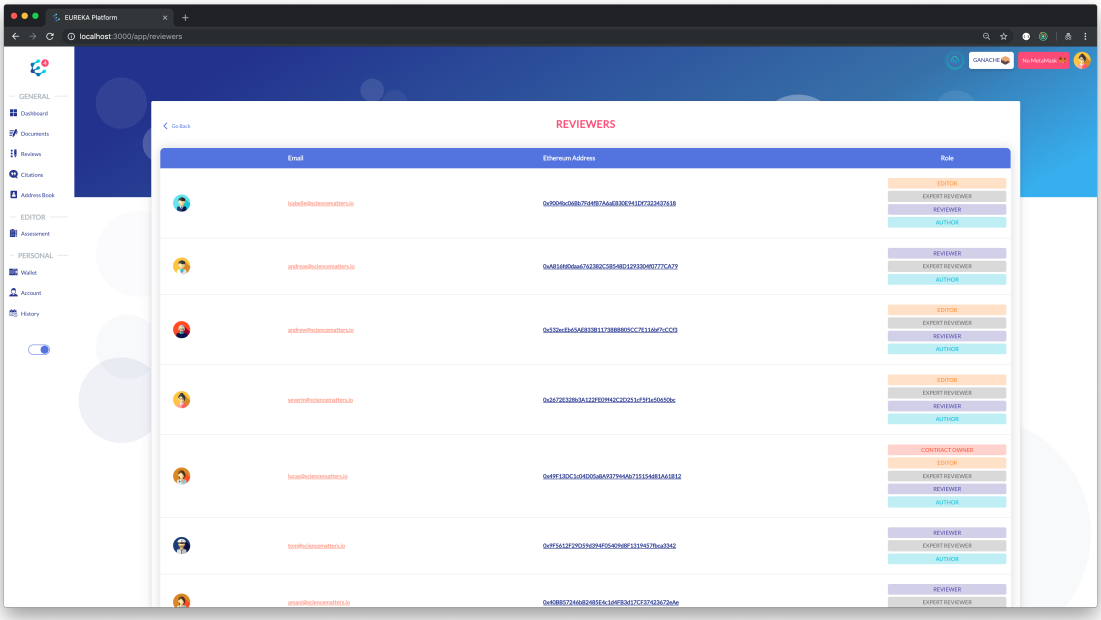



Figure 4.23: Editorial Assessment - Pool of Reviewers

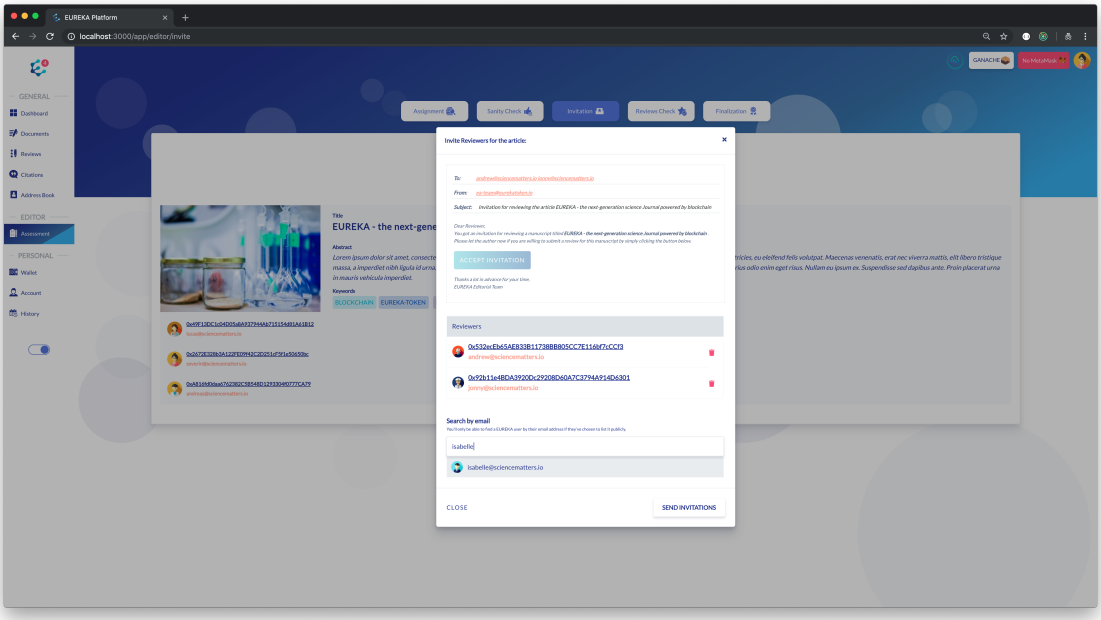



Figure 4.24: Editorial Assessment - Reviewers Invitation

*Reviews Check* Figure 4.25 illustrate the fourth step of the Editorial Assessment process. In this step, editors can browse all handed-in reviews and state whether they are sensible or not.

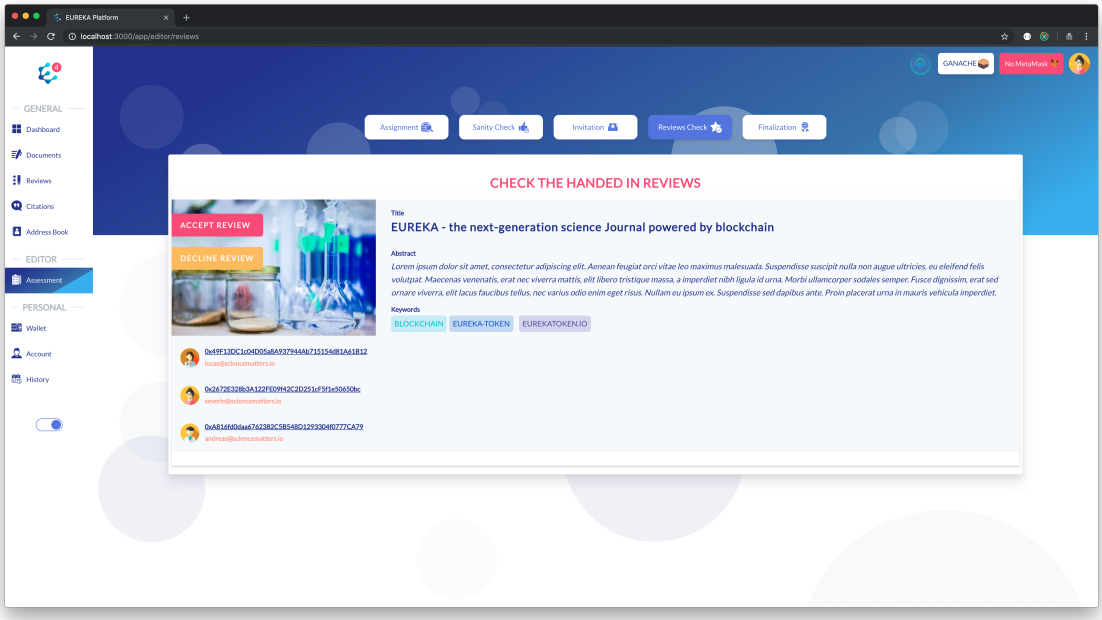



Figure 4.25: Editorial Assessment - Reviews Check

*Finalization* Figure 4.26 illustrate the last step of the Editorial Assessment process. In this step, editors can finalize the articles and trigger consequently the token-based reward distribution.

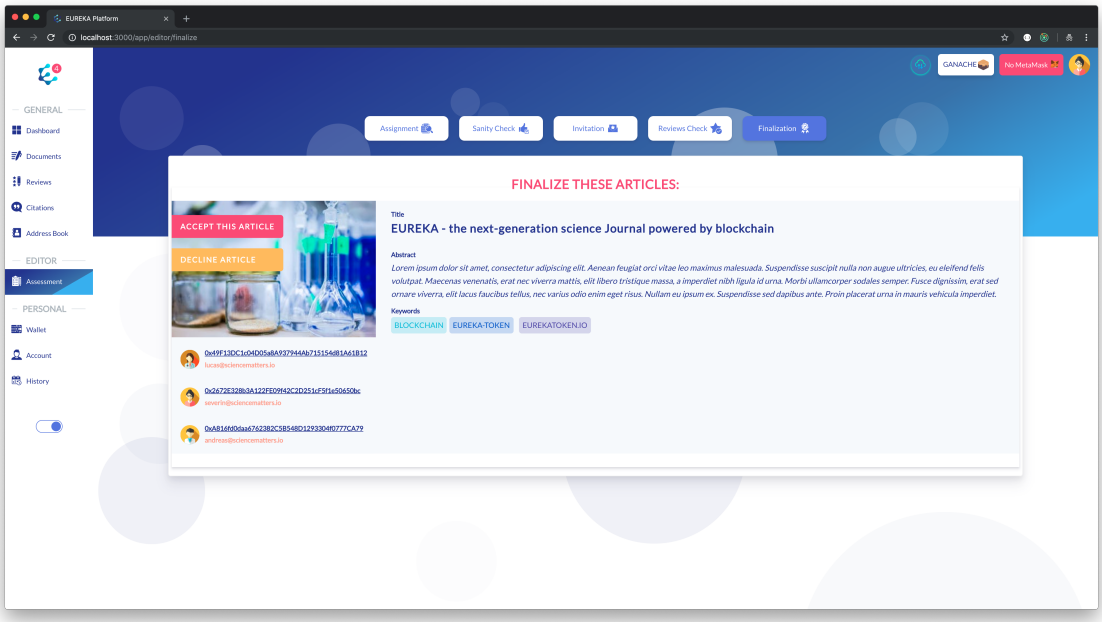



Figure 4.26: Editorial Assessment - Article Finalization

### Peer-Review Process

Figure 4.27 and 4.28 show the peer-review process. Reviewers can accept review invitations and are able to annotate articles directly in the browser and submit their review.

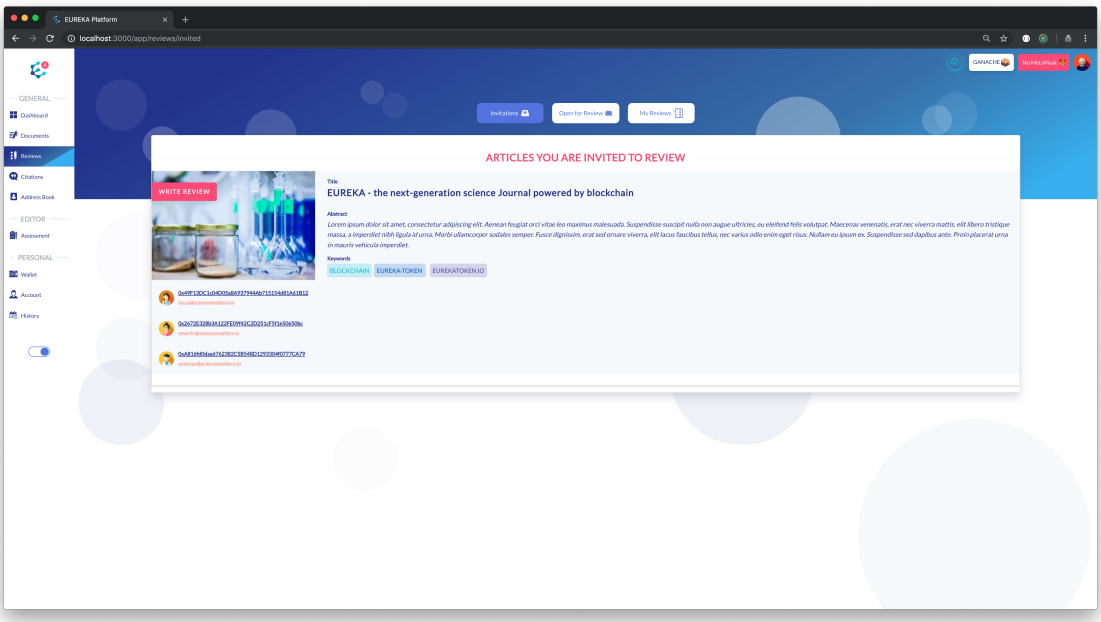



Figure 4.27: Peer Review - Invitation

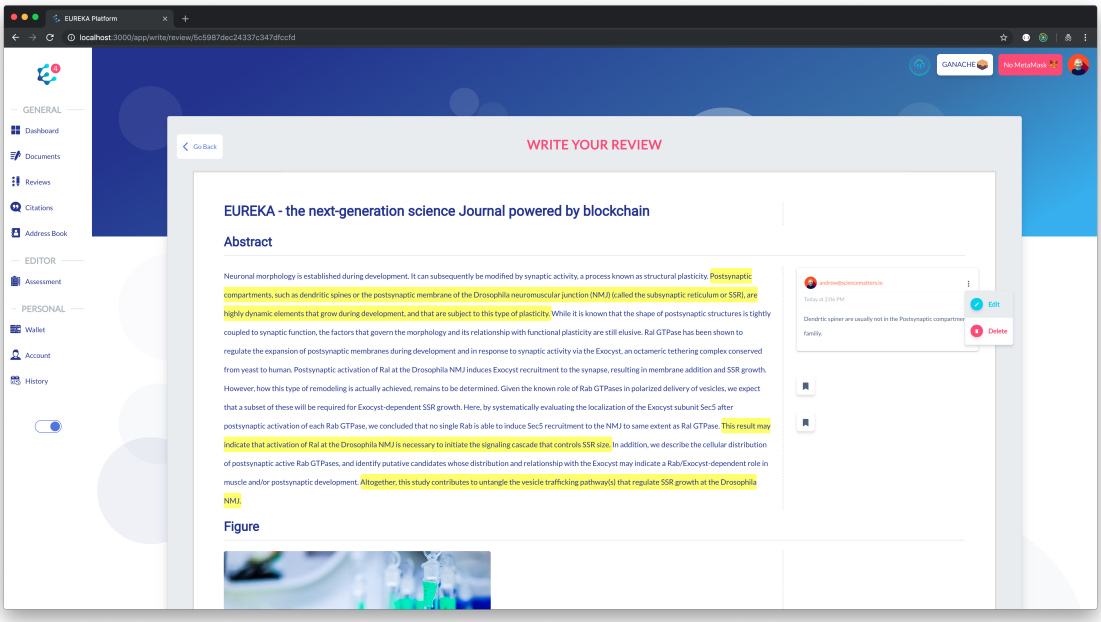



Figure 4.28: Peer Review - Annotation Tool

### User Profile

Figure 4.29 illustrates the User Profile screen. The different EKA Token and Ether amounts are displayed below the User avatar.

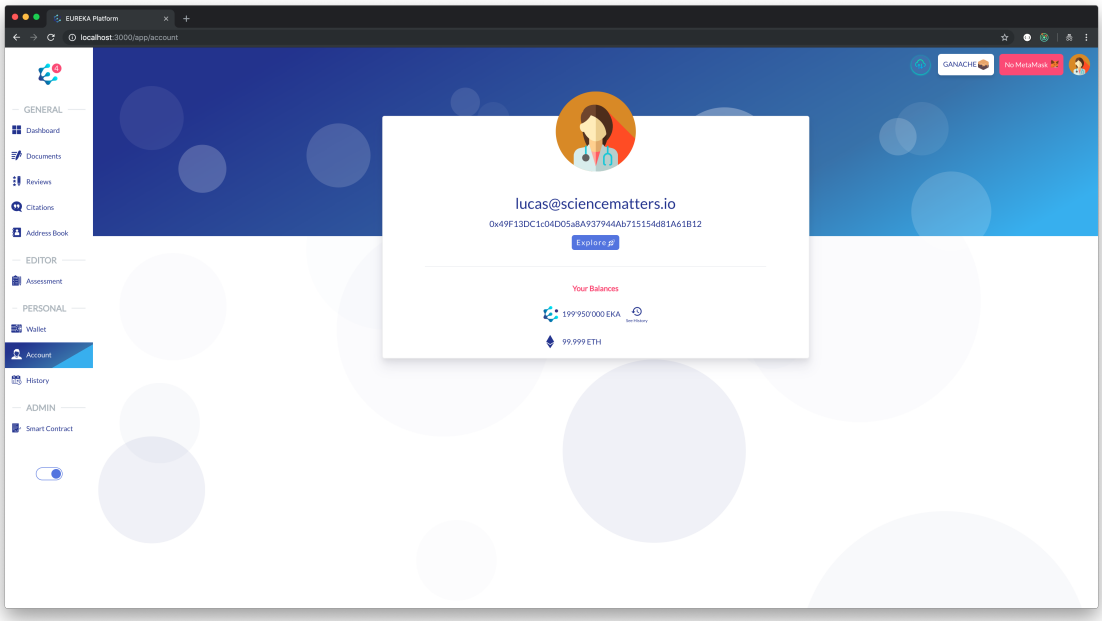



Figure 4.29: User Account

### Blockchain History

Figure 4.30 illustrates the history of all transaction that have been performed by the user and simultaneously recorded on the Ethereum blockchain. The user is able to browse them and retrieve different transaction-related data, such timestamps or Ethereum addresses.

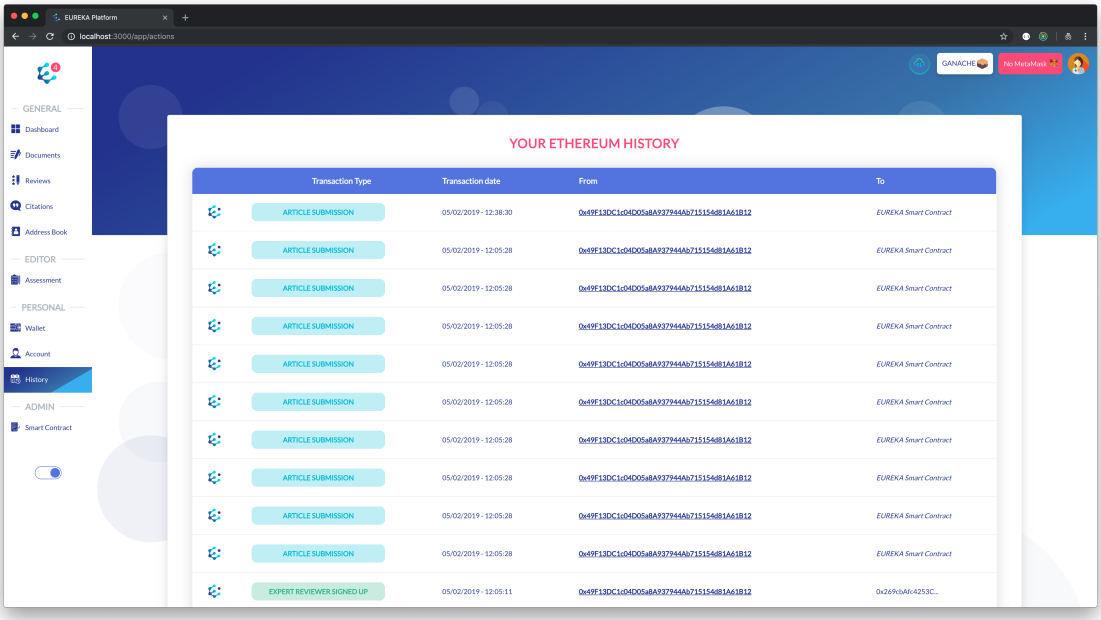



Figure 4.30: Blockchain Transaction History

## Smart Contract Owner Dashboard

Figure 4.31 illustrates SC dashboard. The SC Owner is able to perform some protected methods, such as assigning expert reviewers ,described in the SC using this Dashboard.

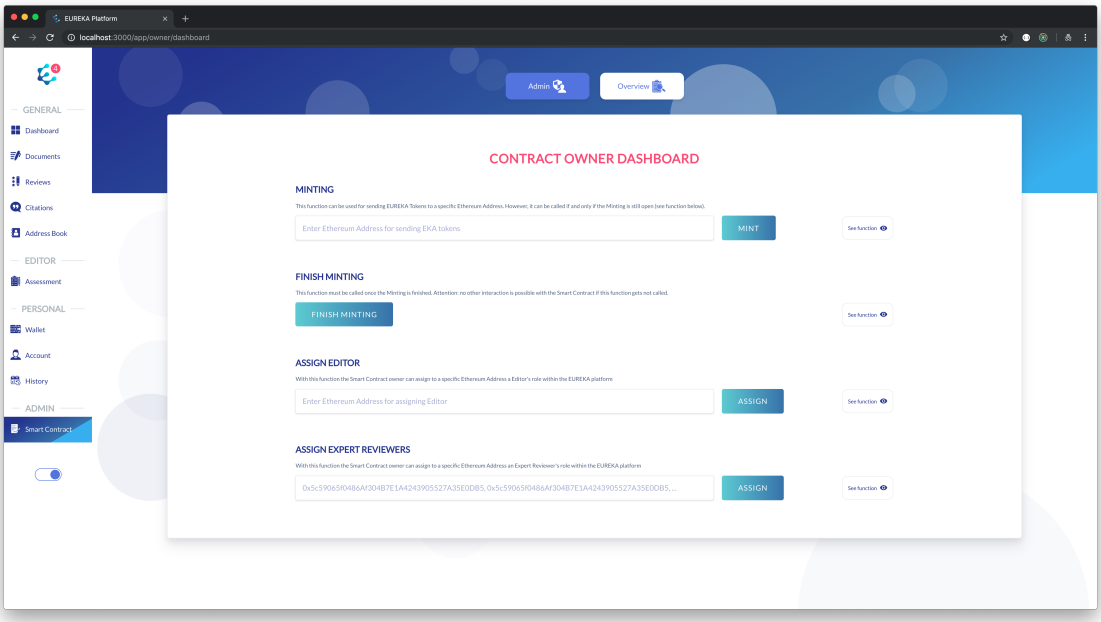



Figure 4.31: Smart Contract Owner Dashboard

### Mobile Responsiveness

Figure 4.32, 4.33, 4.34, illustrates the Mobile Responsiveness of the EUREKA platform. Indeed, all screens have been designed taking into consideration different devices dimensions.

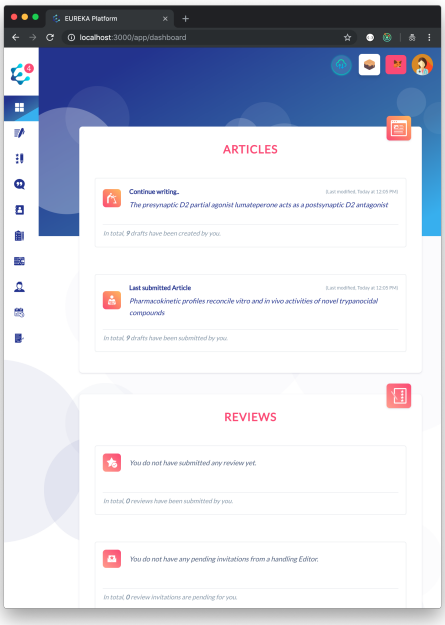



Figure 4.32: Mobile Version - Dashboard

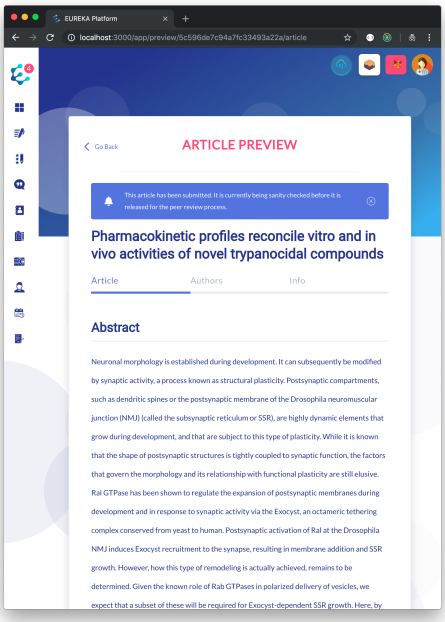



Figure 4.33: Mobile Version - Article Preview

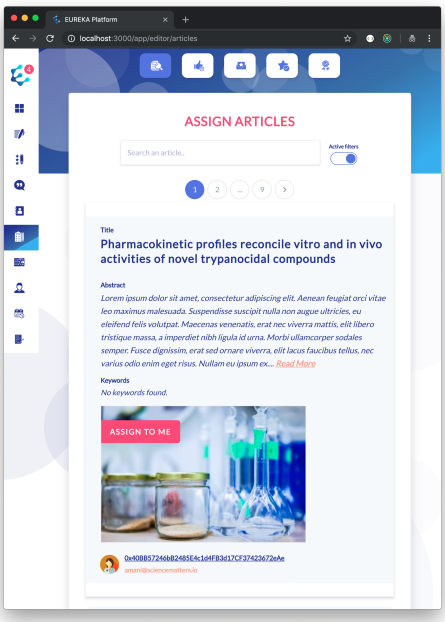



Figure 4.34: Mobile Version - Editorial Assessment - Article Assignment

## 4.4 Smart contracts

The Smart Contracts are written in Solidity v0.4.25 [24]. The official documentation of Solidity states the following:

> Solidity is a contract-oriented, high-level language for implementing smart contracts. It was influenced by C++, Python and JavaScript and is designed to target the Ethereum Virtual Machine (EVM). [24]

In the future the EUREKA ecosystem will contains one so called *EUREKA Token Contract* and several *Journal Contracts*. It will be possible to customize the Journal Contract by parameterizing it and to deploy a new contract from within the EUREKA platform.

### 4.4.1 Token contract

The EUREKA Token Contract is an ERC-20 token [11]. It is a standard for implementing smart contracts on the Ethereum blockchain. In contains all the relevant methods and variables for representing a token respectively a crypto-currency. It implements the usual functions like transfer, transferFrom, totalSupply balanceOf etc. and the parameters token name, symbol and decimals are set to ”EUREKA token”, ”EKA” and 18 decimals. Although the token contract is not yet deployed to the main network and mined, the maximum supply is planned to be set to 298’607’040.

The EUREKA Token Contract also contains parts of more extending smart contract standards which are all sub standards of the ERC-20 token. Specifically we have a transferAndCall token interface which deviates slightly from the ERC-677 token [12] since we only implement a tranferAndCall method and no tokenFallback method. The tokenFallback method was left out because we implemented the transferAndCall method in a generic way such that the method which wants to be called can be specified as a method parameter.

```
1  // ERC677 functionality
2  function transferAndCall(address _to, uint256 _value, uint8 _rewardType,
       bytes4 _methodName, bytes _args) public returns (bool) {
3
4    require(mintingDone == true);
5    require(transfer(_to, _value, _rewardType));
6
7    emit Transfer(msg.sender, _to, _value, _methodName, _args);
8
9    // call receiver
10   if (Utils.isContract(_to)) {
11     require(_to.call(_methodName, msg.sender, _value, _args));
12   }
13   return true;
14 }
```

Listing 4.1: The generic transferAndCall method of the EUREKA token contract

The EUREKA Token Contract also implements the ERC-865 interface [13]. It represents the implementation of a delegated transfer method which results in a so called tranferPreSigned method. It is the prerequisite for the payment service which allows to pay the transaction costs (gas) in EKA tokens instead of Ether.

We also extend the standard ERC-865 token by implementing not only a delegated transfer method, we also provide a delegated transferAndCall method (transferAndCallPreSigned) to assure that the user can call every method after transferring without having Ether in his wallet.

```
1  function transferPreSigned(bytes _signature, address _to, uint256 _value
       , uint256 _fee, uint256 _nonce, uint8 _fromType) public returns (bool
       ) {
2
3    require(signatures[_signature] == false);
4
5    bytes32 hashedTx = Utils.transferPreSignedHashing(address(this), _to,
       _value, _fee, _nonce);
6    address from = Utils.recover(hashedTx, _signature);
7    require(from != address(0));
8
9    doTransfer(from, _to, _value, _fee, msg.sender, _fromType);
10   signatures[_signature] = true;
11
12   emit Transfer(from, _to, _value);
13   emit Transfer(from, msg.sender, _fee);
14   emit TransferPreSigned(from, _to, msg.sender, _value, _fee);
15   return true;
```

```
16 }
```

Listing 4.2: The transferPreSigned method which is implemented by the ERC-865 token standard interface [13]

### 4.4.2 Journal contracts (platform contract)

There exists only one journal contract at the moment. That is why it is often, also in the code and from now on referred as (EUREKA) platform contract.

The object model of the platform contract can be seen in Figure 4.35. The smart contract contains three so called structs [25] which can be seen as "Solidity-Objects".

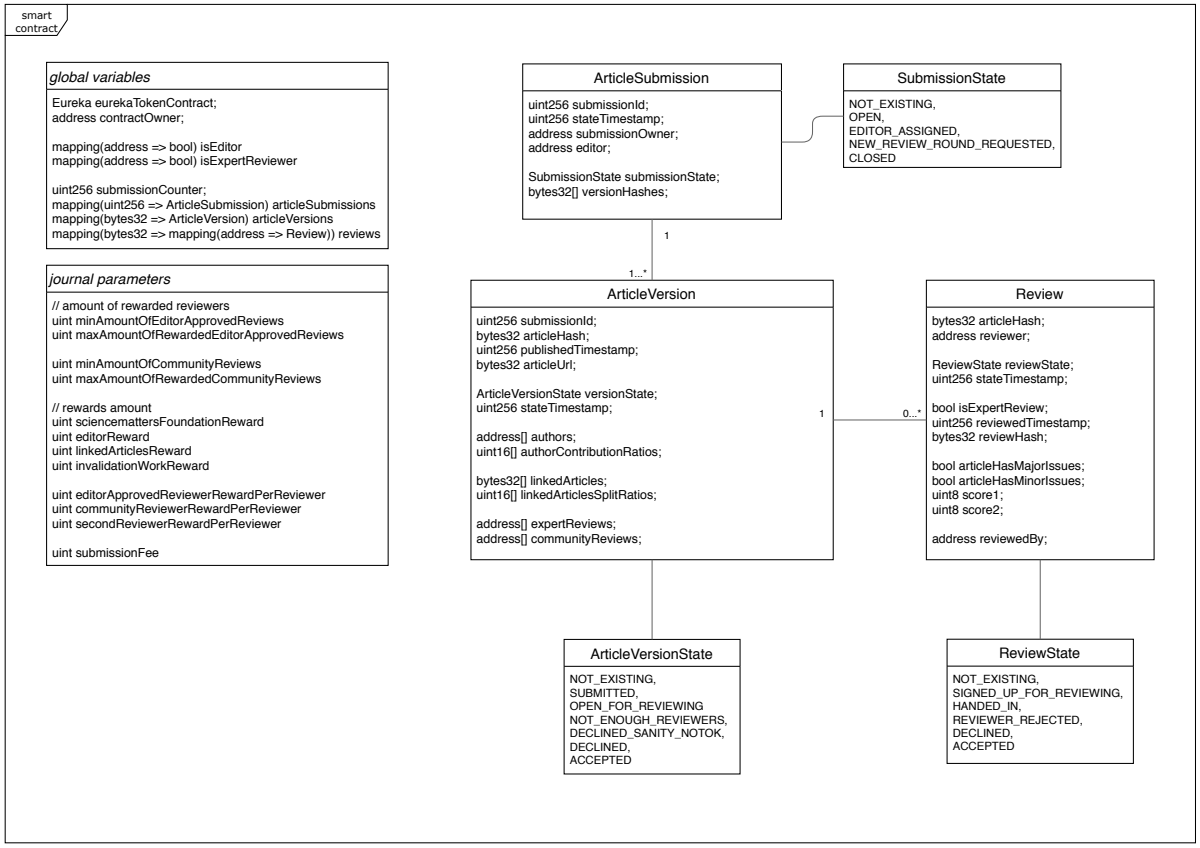



Figure 4.35: Object model of the current version of the platform contract

The main struct is the ArticleSubmission which represents the submission process object. It contains a submissionOwner and a (handling) editor which is assigned to that submission process. And most importantly the ArticleSubmission struct contains the article versions of a submission process. An ArticleSubmission object is created when the first ArticleVersion is created and can theoretically contain as many article versions as the data type allows. Within the ArticleSubmission struct only a bytes32 array with the article version hashes is saved. The ArticleVersion structs are saved globally in a `mapping(bytes32 => ArticleVersion)`. The various ArticleSubmission processes are

stored in a global smart contract mapping as well. The mapping takes the submissionId as a key: `mapping(uint256 => ArticleSubmission)`.

The ArticleVersion struct contains besides a reference to the ArticleSubmission struct (submissionId) all the meta data relevant for the submitted article. The most important variable is the articleHash: All the fields of the submitted documents which is saved on the database within the convenience layer are hashed separately, then concatenated always in the same order and hashed again. In that way, we always have the same unique hash for the same document. It is also possible to hash the PDF of an article if an author decides to not use our in-platform article editor. Also the URL where the article is stored (articleUrl) and the time stamp of the publishing time is saved. To reward authors if they are referenced in a later stage, an array of Ethereum addresses of the authors is stored in the ArticleVersion struct. To be able to define the workload respectively the impact of each author on the article, an authorContributionRatios array is stored in the struct in which the user can define how the contribution is distributed (e.g. [3334, 3333, 3333] for three authors contributed in equal parts). The linked article references are handled the same way as the authors: also a ratio can be specified.

```
1  enum ArticleVersionState {
2    NOT_EXISTING,
3    SUBMITTED,
4    OPEN_FOR_ALL_REVIEWERS,
5    NOT_ENOUGH_REVIEWERS,
6    DECLINED_SANITY_NOTOK,
7    DECLINED,
8    ACCEPTED
9  }
10
11 // an ArticleSubmission can have different versions
12 struct ArticleVersion {
13   uint256 submissionId;
14   bytes32 articleHash;
15   // the timestamp when the article was published
16   uint256 publishedTimestamp;
17   // the URL where the article is saved
18   bytes32 articleUrl;
19
20   ArticleVersionState versionState;
21   uint256 stateTimestamp;
22
23   address[] authors;
24   // the submission owner can weight the contributions of the different
        authors [0;10000]
25   // ( e.g. 3 authors with 1/3 contribution each: {3334,3333,3333} )
26   uint16[] authorContributionRatios;
27   // the hashes of the linked articles
28   bytes32[] linkedArticles;
29   // the submission owner can weight the impact of the linked articles
        [0;10000]
30   uint16[] linkedArticlesSplitRatios;
31
32   // the reviewers which are approved from the editor
33   address[] expertReviews;
34   // every community reviewer can add a community review without being
```

```
        approved
35    address[] communityReviews;
36  }
```

Listing 4.3: The article version struct and its state enum

The Review struct is structured similar then ArticleVersion. Not all review elements are stored on the blockchain either. Only a reviewHash of the data and meta data is saved to the smart contract. Besides the reviewedTimestamp the meta data includes what type of review it is, the scores of the review and if it only includes minor or also major issues. If a review declares major issues in the article and this review is accepted, then the editor is not able to accept the article. The Ethereum address of the user who checks the review for his validity is saved to the reviewedBy variable. The review contains a reference to the ArticleVersion it belongs to and the reviewers address is not only saved in the struct itself it is also saved to an address array in the ArticleVersion struct (expertReviews or communityReviews).

Every struct has its state which is saved in an enum struct. This enum object includes every state which the struct can be in. With every state change the time when the change happened is stored. With this stateTimestamp variable it is possible to check the time which for example a reviewer has already used to review an article. If this time is too large, a state change can be triggered. The functionalities of the time based smart contract and the according monitor can be seen in section 4.7.

The platform contract contains several global variables which will be definable as constructor parameters during the deployment of a journal contract. At the moment these variables are just set to a specific value. The platform contract monitor from the convenience layer pulls the journal parameters by calling the getJournalParameters view. The contractOwner is set in the constructor and, for developing purposes, the eurekaTokenContract address can be set by calling the setEurekaTokenContract method. When it is minted in the Ethereum main network the eurekaTokenContract variable will be fixed to the according contract address.

```
1   contract EurekaPlatform {
2
3     using SafeMath for uint256;
4
5     Eureka eurekaTokenContract;
6     address public contractOwner;
7
8     /*
9     *   journal parameters
10    */
11    // amount of rewarded reviewers
12    uint minAmountOfExpertReviews = 2;
13    uint maxAmountOfRewardedExpertReviews = 3;
14    uint minAmountOfCommunityReviews = 0;
15    uint maxAmountOfRewardedCommunityReviews = 10;
16
17    // rewards amount
18    uint sciencemattersFoundationReward = 1250;
19    uint editorReward = 500;
20    uint linkedArticlesReward = 750;
```

```
21   uint invalidationWorkReward = 1000;
22   uint expertReviewerRewardPerReviewer = 250;
23   uint communityReviewerRewardPerReviewer = 50;
24   uint secondReviewerRewardPerReviewer = 25;
25
26   // resulting submissionFee
27   uint public submissionFee =
28   sciencemattersFoundationReward
29   + editorReward
30   + linkedArticlesReward
31   + invalidationWorkReward
32   + maxAmountOfRewardedExpertReviews * expertReviewerRewardPerReviewer
33   + maxAmountOfRewardedCommunityReviews *
        communityReviewerRewardPerReviewer
34   + maxAmountOfRewardedCommunityReviews *
        secondReviewerRewardPerReviewer;
35
36   uint constant maxReviewRounds = 3;
37
38
39   constructor(address _eurekaTokenContractAddress) public {
40
41     eurekaTokenContract = Eureka(_eurekaTokenContractAddress);
42     contractOwner = msg.sender;
43   }
44
45   //this method is only needed in dev environment
46   function setEurekaTokenContract(address _eurekaTokenContractAddress)
        public {
47
48     require(msg.sender == contractOwner, "only the contract owner can
          call this function");
49     eurekaTokenContract = Eureka(_eurekaTokenContractAddress);
50   }
51
52   //... rest of the platfrom contract
53
54 }
```

Listing 4.4: The first part of the platform contract; the global smart contract variables including all journal parameters

Besides the three global mappings which save articleSubmissions, articleVersions and reviews the platform contract contains two other mappings, namely the isEditor and the isExpertReviewer mapping (both: `mapping(address => bool)`). The journal respectively the contract owner can call the signUpEditor function or the signUpExpertReviewer function to assign the according role to an Ethereum address. The booleans in the mapping will be set to `true` for the specified address. A user can only be editor or expert reviewer if his address is set to true by the contract owner.

```
55
56 /* primary key mappings */
57
58 uint256 submissionCounter;
59 mapping(uint256 => ArticleSubmission) public articleSubmissions;
60 mapping(bytes32 => ArticleVersion) public articleVersions;
```

```
61  // mapping by articleHash and reviewer address
62  mapping(bytes32 => mapping(address => Review)) public  reviews;
63
64
65
66  /* journal administration methods */
67
68  event EditorSignUp(...);
69  function signUpEditor(address editor) public {
70
71    require(msg.sender == contractOwner);
72
73    isEditor[editor] = true;
74    emit EditorSignUp(...);
75  }
76
77  event EditorResigned(...);
78  function resignEditor(address editor) public {
79
80    require(msg.sender == contractOwner);
81
82    isEditor[editor] = false;
83    emit EditorResigned(...);
84  }
85
86  event ExpertReviewerSignUp(...);
87  function signUpExpertReviewer(address expertReviewer) public {
88
89    require(msg.sender == contractOwner);
90
91    isExpertReviewer[expertReviewer] = true;
92    emit ExpertReviewerSignUp(...);
93  }
94
95  function signUpExpertReviewers(address[] expertReviewers) public {
96
97    require(msg.sender == contractOwner);
98
99    for(uint i=0; i < expertReviewers.length; i++) {
100     isExpertReviewer[expertReviewers[i]] = true;
101     emit ExpertReviewerSignUp(...);
102   }
103 }
104
105 event ExpertReviewerResigned(...);
106 function resignExpertReviewer(address expertReviewer) public {
107
108   require(msg.sender == contractOwner);
109
110   isExpertReviewer[expertReviewer] = false;
111   emit ExpertReviewerResigned(...);
112 }
```

Listing 4.5: The ”role-mappings”, different roles can be assigned to Ethereum addresses

As already seen in the code snippet above (listing 4.4), it is used SafeMath [14] to securely

handle multiplication and especially division. SafeMath is for example used when the specified rewards for reviewers is divided by the number of review rounds since it it cannot be assured that the reviewer from the first round is also answering in the second review round. (See line 52 in the code snippet in section 4.4.3.)

### 4.4.3 Smart contract interactions

In this section the most relevant methods of the smart contracts are described.

#### Creating a submission process by submitting an article to be peer reviewed

The submission process starts by calling the EUREKA token contract's transferAndCall method. Belows listing 4.6 shows the web3 interface method for calling the token contract. The transferAndCall method transfers the specified _amount of EKA tokens to the address from the first method parameter (_to). If the transfer is successful, the method checks if the _to address is a valid contract address and calls its method specified in the _methodName parameter. As seen in the transferAndCall method in listing 4.1 the article document _data is passend to the platform contract via _args parameter.

```
1 const submitArticle = (_contract, _to, _amount, _data) => {
2   return _contract.methods
3     .transferAndCall(
4       _to,
5       _amount,
6       '0x0861681c',
7       _data
8     );
9 };
```

Listing 4.6: The web3 interface method for submitting an article which calls the token contract's transferAndCall method

As one can see in listing 4.6 for submitting an article the method with the method ID "0x0861681c" is called. The stated ID represents the startSubmissionProcess method from the platform contract (listing 4.7). These are the first four bytes of the keccak hash of the method name including the parameter types [26].

First it is checked if the message sender is the EUREKA token contract, everything else is not allowed. After that, it is required that the transfered money is equal the specified submission fee in the journal contract. In that way, it is made sure that it is possible to have different journals with the same token contract since we check the amount in the journal contract and not in the token contract. Besides setting the meta data (such as the submission owner) and setting the submission state to OPEN, also the submitArticleVersion method is triggered where the first article version struct is created and saved to the articleVersions array of the articleSubmission struct. Also the article version is timestamped by setting the publishedTimestamp variable to the block.timestamp time (see line 16 in listing 4.8).

```
function startSubmissionProcess(
  address _from,
  uint256 _value,
  bytes32 _articleHash,
  bytes32 _articleURL,
  address[] _authors,
  uint16[] _authorContributionRatios,
  bytes32[] _linkedArticles,
  uint16[] _linkedArticlesSplitRatios) public {


  require(msg.sender == address(eurekaTokenContract), 'the method call
     needs to come from the EUREKA token contract');
  require(_value == submissionFee, 'transferred amount needs to be equal
      to the submission fee');

  uint submissionId = submissionCounter++;
  ArticleSubmission storage submission = articleSubmissions[submissionId
     ];

  submission.submissionId = submissionId;
  submission.submissionOwner = _from;

  submitArticleVersion(submissionId, _articleHash, _articleURL, _authors
     , _authorContributionRatios, _linkedArticles,
     _linkedArticlesSplitRatios);

  submission.submissionState = SubmissionState.OPEN;
  submission.stateTimestamp = block.timestamp;

  emit SubmissionProcessStart(...);
}
```

Listing 4.7: The platform contract method of starting a submission process

```
function submitArticleVersion(
  uint256 _submissionId,
  bytes32 _articleHash,
  bytes32 _articleURL,
  address[] _authors,
  uint16[] _authorContributionRatios,
  bytes32[] _linkedArticles,
  uint16[] _linkedArticlesSplitRatios) private {

  ArticleVersion storage article = articleVersions[_articleHash];
  require(article.submissionId == 0, 'Article was already uploaded.');
    // edge case: two articles have the same hash

  article.submissionId = _submissionId;
  article.articleHash = _articleHash;
  article.articleUrl = _articleURL;
  article.publishedTimestamp = block.timestamp;

  article.authors = _authors;
  article.authorContributionRatios = _authorContributionRatios;
  article.linkedArticles = _linkedArticles;
  article.linkedArticlesSplitRatios = _linkedArticlesSplitRatios;
```

```
23
24    articleSubmissions[_submissionId].versions.push(article.articleHash);
25    article.versionState = ArticleVersionState.SUBMITTED;
26    article.stateTimestamp = block.timestamp;
27  }
```

Listing 4.8: The platform contract method of creating an article version and timestamping it

**A handling editor assigns himself to the submission process (representative for all simple state change interactions)**

These two steps of the peer-review process are quite simple. Listing 4.9 shows the assignForSubmissionProcess method which is representative for every state changing method. Every function is set up similarly. First the global requirements are checked, then the struct is pulled and struct-dependent requirements are checked too. In that way, it is made sure that the function is only doing the changes, when global variables and struct are in the correct state. After the checks the struct changes are performed (see listing 4.9, line 13-15) and in the end event is emit to notify all listeners that the smart contract state has changed.

```
1  function assignForSubmissionProcess(uint256 _submissionId) public {
2
3    require(isEditor[msg.sender], "the sender of the transaction must be
       an editor to call this function.");
4
5    ArticleSubmission storage submission = articleSubmissions[
       _submissionId];
6
7    // different requirements checks
8    require(submission.submissionState == SubmissionState.OPEN, "the
       submission process needs to be OPEN to call this method.");
9    require(submission.editor == address(0), "the submission process is
       already assigned to an editor.");
10   require(msg.sender != submission.submissionOwner, "the submission
       owner can't be the editor of the same submission process");
11
12   // perform changes on ArticleSubmission struct
13   submission.editor = msg.sender;
14   submission.submissionState = SubmissionState.EDITOR_ASSIGNED;
15   submission.stateTimestamp = block.timestamp;
16
17   emit AssignmentForSubmissionProcess(...);
18 }
```

Listing 4.9: The assignForSubmissionProcess method saves an address to the submission struct's variable "editor"

### An expert reviewer submits his review

When an expert reviewer (in earlier versions expert reviewers were called editor approved reviewers) wants to review an article, he can either sign up for the review (an extra transaction is sent to the smart contract to reserve the reward) or skip this step and directly submit the review. Listing 4.10 shows the function which is called by the reviewer to add a review respectively the meta data and the hash of the review. In line 17 and 18 the function checks if the review already exists (it exists already if the reviewer has signed up for the review). If not the signUpForReviewing function (listing 4.11) is called first to create the review struct.

```
1  function addExpertReview(
2    bytes32 _articleHash,
3    bytes32 _reviewHash,
4    bool _articleHasMajorIssues,
5    bool _articleHasMinorIssues,
6    uint8 _score1,
7    uint8 _score2) public {
8
9    require(isExpertReviewer[msg.sender], "msg.sender must be an expert
       reviewer to add an expert review.");
10   require(articleVersions[_articleHash].versionState ==
       ArticleVersionState.OPEN_FOR_ALL_REVIEWERS, "this method can't be
       called. version state must be OPEN_FOR_ALL_REVIEWERS.");
11
12   Review storage review = reviews[_articleHash][msg.sender];
13
14   require(review.reviewState <= ReviewState.SIGNED_UP_FOR_REVIEWING,
15     "msg.sender cannot add an expert review");
16
17   if (review.reviewState == ReviewState.NOT_EXISTING)
18     signUpForReviewing(_articleHash);
19
20   review.isExpertReview = true;
21   review.reviewHash = _reviewHash;
22   review.reviewedTimestamp = block.timestamp;
23   review.articleHasMajorIssues = _articleHasMajorIssues;
24   review.articleHasMinorIssues = _articleHasMinorIssues;
25   review.score1 = _score1;
26   review.score2 = _score2;
27
28   review.reviewState = ReviewState.HANDED_IN;
29   review.stateTimestamp = block.timestamp;
30
31   emit ExpertReviewIsAdded(...);
32 }
```

Listing 4.10: addExpertReview function which will save the meta data of the review and its hash

```
1  function signUpForReviewing(bytes32 _articleHash) public {
2
3    require(isExpertReviewer[msg.sender], "msg.sender must be an expert
       reviewer to sign up for adding an expert review.");
4
```

```
5   require(articleVersions[_articleHash].versionState ==
        ArticleVersionState.OPEN_FOR_ALL_REVIEWERS, "this method can't be
        called. version state must be OPEN_FOR_ALL_REVIEWERS.");
6
7   Review storage review = reviews[_articleHash][msg.sender];
8
9   require(review.reviewState == ReviewState.NOT_EXISTING, "the review
        already exists or the reviewer is already signed up.");
10
11  // save the reviewer data
12  review.reviewState = ReviewState.SIGNED_UP_FOR_REVIEWING;
13  review.stateTimestamp = block.timestamp;
14  review.reviewer = msg.sender;
15  review.isExpertReview = true;
16
17  //save the address of the reviewer to the articleVersion struct
18  articleVersions[_articleHash].expertReviews.push(review.reviewer);
19
20  emit SignedUpForReviewing(...);
21 }
```

Listing 4.11: The signUpForReviewing function which creates the review struct and reserves the reward for a specific amount of time

### The handling editor checks the handed in reviews and finalizes the article submission process

As soon as reviews are handed in someone needs to check them. Currently only the editor can accept or decline reviews. The editor checks expert reviews as well as community reviews. For a final version it is planed that the editor checks the expert reviews only and community reviews can either be checked by expert reviews or more generally by reviewers with a good enough score (not implemented yet).

In listing 4.12 one can see, that the changes made to the review state are very small. Only the state is changed and the variable reviewedBy is filled with msg.sender (the Ethereum address which has sent the transaction). If a review is declined, the reviewer has the chance to correct its review. The corrected review needs to be checked by the same address as it was checked in the first round.

```
1  function acceptReview(
2    bytes32 _articleHash,
3    address _reviewerAddress) public {
4
5    // in the future not only the editor will be able to accept reviews.
         for example expert reviewers or reviewers with a high reputation
         score will be able to do it
6    require(articleSubmissions[articleVersions[_articleHash].submissionId
         ].editor == msg.sender, "msg.sender must be the editor of this
         submission process");
7
8    Review storage review = reviews[_articleHash][_reviewerAddress];
9
```

```
10    require(review.reviewState == ReviewState.HANDED_IN, "to accept the
          review, the review state must be HANDED_IN.");
11
12    require(review.reviewedBy == address(0)
13    || review.reviewedBy == msg.sender, "a corrected review must be
          checked from the same user than before.");
14
15    review.reviewState = ReviewState.ACCEPTED;
16    review.stateTimestamp = block.timestamp;
17    review.reviewedBy = msg.sender;
18
19    emit ReviewIsAccepted(...);
20  }
```

Listing 4.12: The acceptReview function which can be called by the handling editor

When all reviews are checked and enough reviews are accepted, the article submission process can be finalized. Listing 4.13 shows the acceptArticleVersion function. The biggest part of this functions are requirement checks if the method can be called at all. It includes checks such as if enough reviews were handed in, if enough reviews are accepted or if someone is still reviewing the article. Only if every requirement is passed, then the changes are called: The article version state is set to `ACCEPTED` and the closeSubmissionProcess method is called (line 25 and listing 4.14)

```
1   function acceptArticleVersion(bytes32 _articleHash) public {
2
3     ArticleVersion storage article = articleVersions[_articleHash];
4
5     require(articleSubmissions[article.submissionId].editor == msg.sender,
          "msg.sender must be the editor of this submission process");
6
7     require(article.versionState <= ArticleVersionState.
          OPEN_FOR_ALL_REVIEWERS, "this method can't be called. the article
          version must be sanity checked and must not be already closed.");
8
9     require(countAcceptedReviews(article.articleHash, article.
          expertReviews) >= minAmountOfExpertReviews,
10    "the article doesn't have enough accepted expert reviews to be
          accepted.");
11    require(countAcceptedReviews(article.articleHash, article.
          communityReviews) >= minAmountOfCommunityReviews,
12    "the article doesn't have enough accepted community reviews to be
          accepted.");
13
14    require(countSignedUpForReviewing(article.articleHash, article.
          expertReviews) == 0,
15    "this method can't be called. there are users still working on reviews
          .");
16
17    require(countReviewsWithMajorIssues(article.articleHash, article.
          expertReviews) == 0,
18    "the article needs to be corrected.");
19    require(countReviewsWithMajorIssues(article.articleHash, article.
          communityReviews) == 0,
20    "the article needs to be corrected.");
21
```

```
22    article.versionState = ArticleVersionState.ACCEPTED;
23    article.stateTimestamp = block.timestamp;
24
25    closeSubmissionProcess(article.submissionId);
26
27    emit ArticleVersionIsAccepted(_articleHash, msg.sender, block.
          timestamp);
28  }
```

Listing 4.13: The acceptArticleVersion function which triggers the closeSubmissionProcess function

### The submission process is closed and the rewards are distributed

The following code snippet (listing 4.14) shows again one of the most important method in the platform contract: the method which closes the submission process and distributes the rewards to the different stakeholders.

First the one time rewards are transfered (line 8-9) and after that the amount of review rounds is calculated. This is used in the following lines to distribute the rewards to the expert (listing 4.15)as well as the community reviewers (listing 4.16). For each accepted review a micro payment transfer is triggered. The amount of EKA to be transfered is determined by dividing the ...RewardPerReviewer by the previously calculated number of review rounds (line 51).

Finally the submission state is set to `CLOSED` and a SubmissionProcessClosed event is emitted.

The rewarding of the linked articles and the invalidation work has not been implemented yet.

```
1  event SubmissionProcessClosed(...);
2
3  function closeSubmissionProcess(uint256 _submissionId) private {
4
5    ArticleSubmission storage submission = articleSubmissions[
         _submissionId];
6
7    // transfer rewards to science matters foundation and editor
8    require(eurekaTokenContract.transfer(contractOwner,
         sciencemattersFoundationReward));
9    require(eurekaTokenContract.transfer(submission.editor, editorReward))
         ;
10
11   // counts how many reviewRounds happened to devide the reward later
12   uint reviewRounds = countDeclinedReviewRounds(_submissionId) + 1;
13
14   //distributes the rewards to all reviewers, for every round a seperate
         transfer
15   for (uint i = 0; i < submission.versions.length; i++) {
16
17     ArticleVersion memory articleVersion = articleVersions[submission.
           versions[i]];
```

```
18      if (articleVersion.versionState == ArticleVersionState.DECLINED
19        || articleVersion.versionState == ArticleVersionState.ACCEPTED) {
20
21        rewardExpertReviews(articleVersion, reviewRounds);
22        rewardCommunityReviews(articleVersion, reviewRounds);
23      }
24    }
25
26    //reward linked articles if article is accepted
27    if (articleVersions[submission.versions[submission.versions.length -
          1]].versionState == ArticleVersionState.ACCEPTED) {
28
29      //TODO: reward linkedArticles authors and invalidation work
30      // check also if time is already up
31    }
32    submission.submissionState = SubmissionState.CLOSED;
33    submission.stateTimestamp = block.timestamp;
34
35    emit SubmissionProcessClosed(block.timestamp, _submissionId);
36  }
```

Listing 4.14: The platform contract method of closing a submission process and calling the methods to reward the stakeholders

```
1   function rewardExpertReviews(
2     ArticleVersion _articleVersion,
3     uint _reviewRounds) private {
4
5     uint rewardedReviewers = 0;
6     for (uint i = 0; i < _articleVersion.expertReviews.length; i++) {
7
8       if (rewardedReviewers < maxAmountOfRewardedExpertReviews) {
9
10        if (reviews[_articleVersion.articleHash][_articleVersion.
              expertReviews[i]].reviewState == ReviewState.ACCEPTED) {
11
12          // transfer reward to expert reviewer
13          require(
14            eurekaTokenContract.transfer(
15              _articleVersion.expertReviews[i],
16              expertReviewerRewardPerReviewer.div(_reviewRounds)
17            )
18          );
19          rewardedReviewers++;
20        }
21      }
22      else
23        return;
24    }
25  }
```

Listing 4.15: The platform contract method of transferring the rewards to the expert reviewers

```
1   function rewardCommunityReviews(
2     ArticleVersion _articleVersion,
3     uint _reviewRounds) private {
```

```
4
5   uint rewardedReviewers = 0;
6   for (uint i = 0; i < _articleVersion.communityReviews.length; i++) {
7     if (rewardedReviewers < maxAmountOfRewardedCommunityReviews) {
8       if (reviews[_articleVersion.articleHash][_articleVersion.
            communityReviews[i]].reviewState == ReviewState.ACCEPTED) {
9         // reward community reviewer
10        require(
11        eurekaTokenContract.transfer(
12        _articleVersion.communityReviews[i],
13        communityReviewerRewardPerReviewer.div(_reviewRounds)
14        ));
15        // reward the reviewer of the community review
16        require(
17        eurekaTokenContract.transfer(
18        reviews[_articleVersion.articleHash][_articleVersion.
              communityReviews[i]].reviewedBy,
19        secondReviewerRewardPerReviewer.div(_reviewRounds)
20        ));
21        rewardedReviewers++;
22      }
23    }
24    else
25      return;
26  }
27 }
```

Listing 4.16: The platform contract method of transferring the rewards to the community reviewers

### 4.4.4 Web3 contract methods interface

For using the smart contract function from the back-end and the front-end very easily a web3 contract methods interface was implemented. The ”Smart Contract” folder where the smart contract files are stored was extended by a ”methods” folder. The ’web3-platform-contract-methods.mjs’ file contains all public platform contract function and the ’web3-token-contract-methods.mjs’ file all public token contract functions. The files can be accessed by the back-end as well as the front-end and we can therefore guarantee a consistent single point of truth interface.

Listing 4.17 shows the addCommunityReview method from the web3 platform contract methods interface. One could ask, what the benefit of such a method is: All available methods from the smart contracts are saved in one file, so the programmer does not have to check the smart contract itself and does not have to care about the web3 dependent syntax (`contract.methods.addCommunityReview`).

The previously mentioned listing 4.6 shows a token contract interface method: the submitArticle method. There it is even more useful since the front-end programmer does not need to add the four bytes string which represents the startSubmissionProcess method (`'0x0861681c'`).

```
1
2  export const addCommunityReview = (
3    contract,
4    _articleHash,
5    _reviewHash,
6    _articleHasMajorIssues,
7    _articleHasMinorIssues,
8    _score1,
9    _score2
10 ) => {
11   return contract.methods.addCommunityReview(
12     _articleHash,
13     _reviewHash,
14     _articleHasMajorIssues,
15     _articleHasMinorIssues,
16     _score1,
17     _score2
18   );
19 };
```

Listing 4.17: The addCommunityReview method in the web3 platform contract methods interface

## 4.5 Back-End

In order to bring the interaction with the SC to a level which is easily understandable and intuitive, we create a convenience layer. The main reason for using an convenience layer is as the name already implies is because of convenience. All modification and action on the SC shall be monitored by the convenience layer, so information can be provided prepared to the end user. As the storing information on the blockchain is expensive in terms of gas consumption and memory, and articles often contain a lot of information (especially images and graphs), not all the data will be stored into the blockchain, but only a hash derived from the article (articleHash). With saving only the hash on it, we can assure minimal data storing on the blockchain (which goes in hand with low transaction cost), but at the same time, we provide a fully traceable prove of publishing a specific article through the on-chain written article hash.
The whole data of an article is saved off-chain in our data-base, which will be served by an interplay of our convenience layer, front-end application and the SC on the blockchain.

### 4.5.1 Technical stack

The convenience layer is created like a typical back-end application. We chose Javascript as the programming language to assure parity with the front-end stack. Particularly the reason to use the same library for interaction with the SC led us to this decision. Using npm [61] as the package manager provides us with a big variety of well established libraries and frameworks.
We use Node.js [27] with the Express web application framework [28]. Node.js is a plat-

form built on Chrome's Javascript runtiem for easily building fast and scalable network applications. Express is a light-weight web application framework, which helps to organize your web-application into an MVC [74] on the server side.The front-end application communicates through a REST-Interface[77] with it.

For the storage of data the back-end application sends data to a database. We decided to setup the database with MongoDB [29], as this technology allows us to adapt data structure quickly. In the other hand we use the library Mongoose [30], which extends the functionalities by schema validation, constraints definition and value inspection. The combination of these two technologies allow us to set limitations and control were we need it in order to assure data integrity and allows us to be flexible in areas where it is necessary.

The back-end needs to monitor the SC to keep track of all interaction happening with the SC. We us the Web3.js library [6] to subscribe on events defined within the SC. Web3.js is a collection of libraries, which allow you to interact with a local or remote ethereum node, using HTTP or IPC connection. As events are triggered by the SC, functions get called in the back-end, which update the state into the database. These functions are called with the help of Callback Promises Events.
Is there an state reached, where users needs to be informed (e.g. a user received an invitation to become an editor-approved reviewer), the back-end sends out E-mails to the corresponding user. For this purpose we use the email delivery service SendGrid [31] and its own package-module Sendgrid/mail [57].
Furthermore, the convenience layer is responsible for initializing various functions on the SC depending on the current time (e.g. closing a potential editor-chosen reviewers slot in a review process, if he is not responding within the given time-frame.) For the time-based trigger process of functions we use the Cron extension for Javascript [58]. Cron is a time-based job scheduler in Unix-like computer operating systems.

### 4.5.2 Security provisions

For a proper and safe authentication process we use Passport [32] as middleware and developed a local authentication strategy. For a secure encryption of the users password we make use of the javascript module Bcrypt [75], which uses the bcrypt-algorithm [76] to assure security and store only the encrypted password on our database. In order to grant proper authorization to the platforms users, we implemented our own SC. Each users holds some roles within the database. Roles, which are coupled with a wider access privilege (e.g. contract-owner, editor) can only be granted by an event coming from the SC. This way we assure correct authorization (e.g. Only a users granted to become an editor by the contract-owner on the SC has also the rights to request infos about articles, which must be inspected by editors.)
Express-session [78] provides us the functionality of working with session, which are stored in our databases. User, which have successfully logged in within our platform are provided with a session id. Sessions expire after a defined amount of time has past. All this assures a safe and secure authentication process for our future platform users.

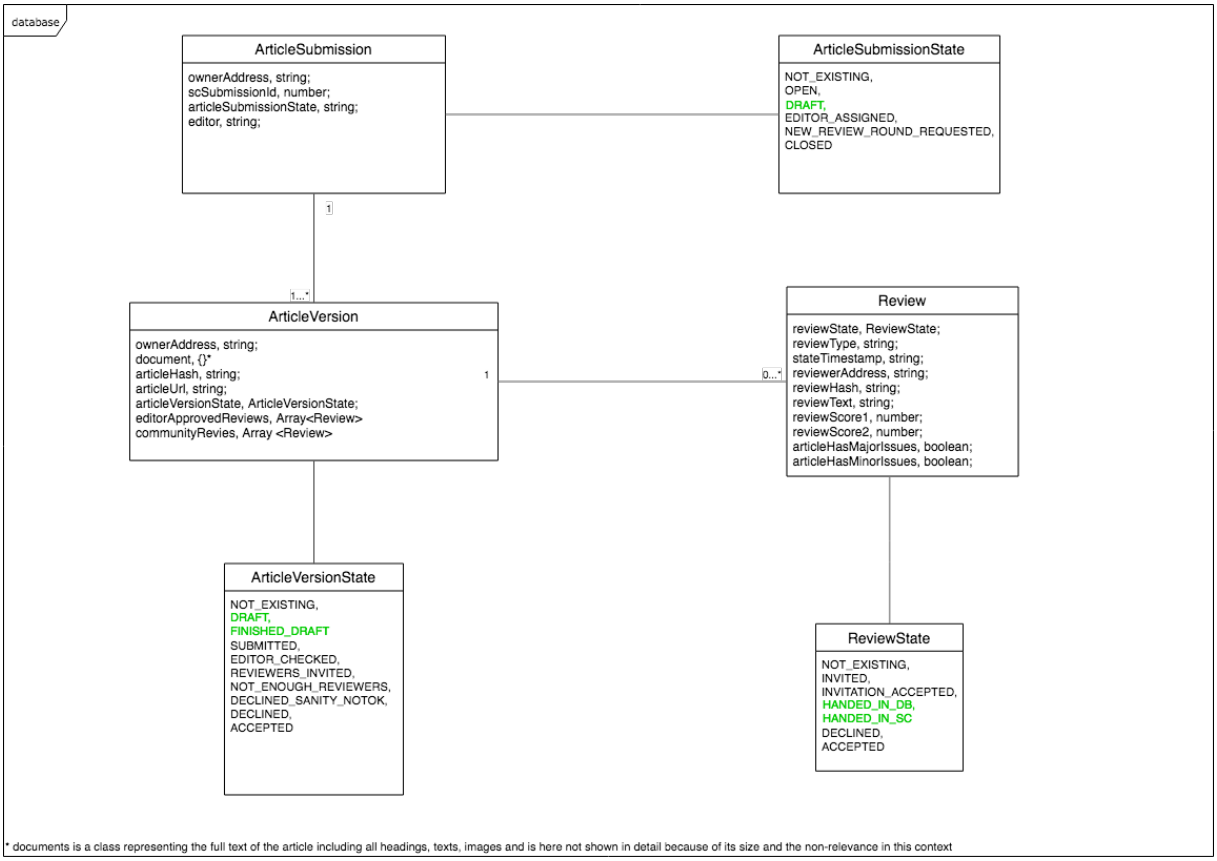



Figure 4.36: Object model of the current version in the data-base

### 4.5.3 Business logic

The main purpose of the convenience layer is to monitor all actions happening on the platform contract and bring interaction with it for a end-user to a intuitive level. It serve as an interconnection between front-end and SC. For that its object-model presented in Figure 4.35 is for the most part mirrored in the database of the back-end. Figure 4.36 shows the object structure on the database.

When an article get first open by an user (the front-end process is more explained in sub-subsection 4.3.1) the front-end request the back-end to create a new articleSubmission object with the ArticleSubmissionState 'DRAFT'. This state exists only in the back-end but not in the SC. Besides that it also already creates the first articleVersion with ArticleVersionState 'DRAFT' At this moment, there has not been any interaction done with the SC therefore no new articleSubmission is pushed to it. When then the user submits its draft to the SC, first a request from the front-end to the back-end updates the ArticleVersionState to 'FINISHED DRAFT'. The second request goes to the SC where the corresponding event gets triggered on the SC-side. The event is monitored by the back-end, which then updates the pre-created articleVersion with the unique articleHash delivered by the SC and also updates the ArticleVersionState to 'SUBMITTED'. The code in Listing 4.37 shows how the data in the back-end gets updated after the event of article submission has been triggered.

```
1       /** Submission Process Start **/
2       EurekaPlatformContract.events.SubmissionProcessStart(
3         undefined,
4         async (error, event) => {
5           if (error) throw error;
6           await articleSubmissionService.
                updateSubmissionStartByArticleHash(
7             event.returnValues.submissionId,
8             event.returnValues.articleHash,
9             event.returnValues.articleURL
10          );
11
12          const additionalInfo = {
13            articleHash: event.returnValues.articleHash,
14            articleURL: event.returnValues.articleURL
15          };
16          await scTransactionService.createScTransaction(
17            event.returnValues.submissionOwner,
18            ScTransactionType.SUBMIT_ARTICLE,
19          event.returnValues.stateTimestamp,
20            event.transactionHash,
21            additionalInfo
22          );
23       }
24      );
```

Figure 4.37: The event-listener for the articleSubmission on the SC back-end side

Similar there are two additional states for the review back-end side compared to the SC. The state turns to 'HANDED_IN_DB' in the back-end, when a reviewer submits its review in the front-end. This fires a request to update the state in the back-end. Again a second request is fired from the front-end to the SC to update it. There an event is triggered, which will let the back-end update the state to 'HANDED_IN_SC'.
With the additional state in the database we can distinct between action triggered in the front-end and those saved within the contract.

### 4.5.4 Structure and responsibilities

This describes the general file structure of the backend source code. Further it shows the overall folder organisation and points out the location of key features and files. Functionalities and features are explained and highlighted wit one of the underlying code examples. Figure 4.38 shows the directory tree listing all important key code files and examples.

```
backend
├── api
├── controller
│   └── access-controller.mjs
├── db
│   ├── article-version-service.mjs
│   ├── user-service.mjs
│   ├── mongoose-db-service.mjs
│   └── .....
├── email
├── helpers
│   ├── bcrypt-hasher.mjs
│   ├── error-thrower.mjs
│   ├── local-passport.mjs
│   └── .....
├── routes
│   ├── index.mjs .4 article-version-routes.mjs
│   ├── user-routes.mjs
│   └── .....
├── schema
│   ├── article-version.mjs
│   ├── user.mjs
│   └── roles-enum.mjs .4 .....
└── web3
    ├── contract-event-listener.mjs
    ├── timebased-contract-service.mjs
    └── .....
```

Figure 4.38: The directory tree of the backend file structure

- **Routes**
  The backend is structured according to the Model-View-Controller (MVC) pattern. There are routes organised in the subfolder 'routes' which define the routing strategy of the Application Programming Interface (API). Figure 4.39 is showing an example of one route handling a GET request for /users/roles. The key routes used for the tokenflow and interaction with the Smart Contract (SC) and frontend are showed in subsection 4.5.5.
  In the subfolder 'db' all services are defined. One service file contains multiple functions all handling some business logic, which mostly interacts with the database in

the background. The route in Figure 4.39 calls the methods 'getAllUsersByRole()' of the 'userService' on Line 10. The 'userservice' handles multiple function containing business logic considering user interaction. In Figure 4.40 the function mentioned above is shown.

```
1
2  router.use(accesController.loggedInOnly);
3  router.get(
4  '/roles',
5  asyncHandler(async req => {
6    if (req.query.role) {
7      if (Roles.hasOwnProperty(req.query.role)) {
8        return await userService.getAllUsersByRole(req.query.role);
9      }
10     return new Error(req.query.role + ' is not a valid role!');
11   }
12   errorThrower.noQueryParameterProvided();
13 })
14 );
```

Figure 4.39: user-routes.mjs: Routing example handling request on '/users/roles'

```
1 import User from '../schema/user.mjs';
2 ....
3 getAllUsersByRole: async role => {
4   return await User.find({roles: role});
5 },
6 ...
```

Figure 4.40: user-service.mjs: Service example managing interaction with the database

- **Services and Schemas**
  A service interacts not directly with the database, but uses mongoose schema, which defines the shape and validation within a MongoDB collection, assuring data quality and consistency in the data base. On line 4 in Figure 4.40 the service calls the schema 'User', which handles interaction with the database.
  A mongoose schema can validate data passed into the database according to user-defined rules. Mostly we validate data by variables it has to contain, correct type of attributes and uniqueness. For each field we also define if it is required or optional. Figure 4.41 shows the schema of a 'User' structure. Line 30 to 35 shows a validation by enumeration for which we again defined allowed values for the attribute 'roles'. Validation by enumeration means that all allowed values of an variables are pre-defined within the schema. These admitted values are described in additional file called 'roles-enum' shown in Figure 4.42. There are further enumerations defined for the state of the article-submission, article-versions and review, as the possible state for all these variables ar defined and limited.

```
import mongoose from 'mongoose';
import Roles from './roles-enum.mjs';

/**
* User of the eureka platform
* Model is used for the local user authentication with passport
*/
const userSchema = mongoose.Schema(
{
  ethereumAddress: {
    type: String,
    required: true,
    unique: true
  },
  password: {
    type: String,
    required: true,
    unique: false
  },
  email: {
    type: String,
    required: true,
    unique: true
  },
  avatar: {
    type: String,
    required: true,
    unique: false
  },
  roles: [
  {
    type: String,
    enum: Object.values(Roles)
  }
  ],
  scTransactions: [{
    type: mongoose.Schema.Types.ObjectId,
    ref: 'ScTransaction'
  }]
},
{collection: 'users'}
);

const User = mongoose.model('User', userSchema, 'users');

export default User;
```

Figure 4.41: user-mjs: Schema example for User collection in the database

```
1  /**
2  * Different roles a platform user can have --> access authorization is
       based on it.
3  */
4  const Roles = Object.freeze({
5    CONTRACT_OWNER: 'CONTRACT_OWNER',
6    ADMIN: 'ADMIN',
7    AUTHOR: 'AUTHOR',
8    REVIEWER: 'REVIEWER',
9    EXPERT_REVIEWER: 'EXPERT_REVIEWER',
10   EDITOR: 'EDITOR',
11   USER: 'USER',
12   ALL: 'ALL'
13 });
14
15 export const getAllRoles = () => {
16   return Object.keys(Roles);
17 };
18
19 export default Roles;
```

Figure 4.42: roles-enum.mjs: Enumeration example used for the User

- **Access-controller**
  Further, we implemented a middle-ware for managing the access to the routes. It is a roles-based access control using the defined roles seen in Figure 4.42. It contains the two functions 'loggedInOnly' (line 17-26) and 'rolesOnly' (line 35-60), both used as middle-wares. The first function checks if a user is logged in when a request is made on one of the underneath routes. If the user is not logged in, it returns an error message (line 21-25), otherwise just let it pass (line 18). An example of usage can be seen in the Route example of Figure 4.39 on line 2. All routes listed below this line can only be accessed when the calling user is logged in, otherwise the error message will be send as response.
  Figure 4.44 is a code snippet from the file 'article-version-routes.mjs' showing the usage of the 'rolesOnly' function. All routes underneath the middle-ware function (line 7) are only accessible by an user, which has the role 'EDITOR'. This presumes that the user must be logged in (Authentication), but also he must have granted the role 'EDITOR' (Authorization). Otherwise the middle-ware will respond with a error message.

- **Helper modules**
  In the sub-folder 'helpers' are helper functions located, which are used generally in various locations in the back-end source code. Starting with 'bcrypt-hasher.mjs', which includes a function called 'hash' which takes a password as parameter and hashes it with the bicrypt hashing function. When a user will provide for example the password 'testWord123', the resolving hash is always exactly '$2a$10$2OovefhQfN5LWL5iEz58xeICNcBuotfAPFre4SJMx/oeALqpNb6cu'. We only save the hash of the password passed into the database. With that we assure even in the event of

```
/**
* Access Controller is handling the authorisation of
* the different REST routes based on the user id.
*/

import User from '../schema/user.mjs';

export default {
  /**
  * Logged-in check used as a middleware function for a router
  * use it like "router.use(authorizationCheck.loggedInOnly)
  * @param req
  * @param res
  * @param next
  * @returns {*}
  */
  loggedInOnly: (req, res, next) => {
    if (req.isAuthenticated()) return next();

    //not logged in
    notLoggedIn(res);
  },

  /**
  * Middleware function, which checks if the user has one of the given
     roles (params)
  * as a role.
  * If he has non he receives a 404 - Not Authorized
  * @param roles as array of string
  * @returns {Function}
  */
  rolesOnly: roles => {
    return function(req, res, next) {
      User.findById(req.user)
      .exec()
      .then(user => {
        let authorized = false;
        if (user.roles.length > 0) {
          for (let i = 0; i < roles.length; i++) {
            if (user.roles.indexOf(roles[i]) >= 0) {
              authorized = true;
            }
          }
          if (authorized) next();
          else {
            notCorrectRoleError(res, roles);
          }
        } else {
        notCorrectRoleError(res, roles);
      }
    })
    .catch(err => {
      res.status(500).json({
        error: err
      });
    });
  };
}
};
```

Figure 4.43: access-controller.mjs: Middleware for handling access control

```
1
2  ...
3
4  /**
5  articles assigned to an Editor
6  **/
7  router.use(accesController.rolesOnly([Roles.EDITOR]));
8  router.get(
9  router.get(
10 '/assigned/signoff',
11 asyncHandler(async req => {
12   let articles = await articleVersionService.getArticlesAssignedTo(
13   req.session.passport.user.ethereumAddress,
14   [ARTICLE_VERSION_STATE.SUBMITTED]
15   );
16   return getArticlesResponse(articles);
17 })
18 );
19
20 ...
```

Figure 4.44: article-version-routes.mjs: Example Code snippet showing the use of rolesOnly

a database break, that passwords can not be stolen, as hashes are not bidirectional resolvable. The second function 'compare' takes a password and a saved hash as parameter returning true if the has of the password matches the pre-saved hash, returning false otherwise.

The file 'error-thrower.mjs' is an accumulation of all error-message inclusive status being thrown by the back-end application in various situation. Figure 4.45 shows some example out of 'error-thrower.mjs', without being complete. The first error thrower (line 3 - 7) is for example used in the access-controller in Figure 4.43 on line 21, calling it, when the user is not authenticated. This file assures consistency of the various error messages being thrown.

For authentication we are using the npm package 'passport' [33] defining our own Local Strategy. As described in 4.3.1 the front-end will provide a password , which is a hash derived from the private key of a user. As username the ethereum address matches perfectly , since it is unique by definition. Figure 4.46 shows the implementation of our own local strategy for authentication. First it checks if the user is in the database (line 22 - 26). If so it checks the password using the bcrypt hasher explain above and compares the deriving hash with the one in the database (line 30 - 34). When these are equal the application responds with a session key, which means a user is considered logged in.

```
...

  notLoggedIn: () => {
    let error = new Error('Access denied, not logged in Backend');
    error.status = 401;
    throw error;
  },
  notCorrectEthereumAddress: () => {
    let error = new Error('Access denied, provided ethereum-address does
        not match owner-address');
    error.status = 403;
    throw error;
  },

  notAuthorizedToDoThisAction: () => {
    let error = new Error('Access denied, the user is not allowed to
       take this action');
    error.status = 403;
    throw error;
  },

  noEntryFoundById: (id) => {
    let error;

    if(id) {
      error = new Error('Could not find the entry with the provided ID:
          ' + id);
    } else {
    error= new Error('Could not find the entry with the provided ID');
  }
  error.status = 400;
  throw error;},
...
```

Figure 4.45: error-thrower.mjs: Example error messages as example code snipped

```
/**
* Local Passport strategy
*/

import passport from 'passport';
import passportLocal from 'passport-local/lib/index';
import bcryptHasher from './bcrypt-hasher.mjs';
import User from '../schema/user.mjs';

const LocalStrategy = passportLocal.Strategy;

passport.use(
new LocalStrategy(
{
  usernameField: 'ethereumAddress',
  passwordField: 'password'
},

async function(ethereumAddress, password, done) {

  // user in db?
  const dbUser = await User.findOne({ethereumAddress: ethereumAddress});
  if(!dbUser) {
    let error = new Error('Provided Login credentials are wrong');
    error.status = 401;
    return done(null, false);
  }

  // correct pwd?
  const isCorrectHash = await bcryptHasher.compare(password, dbUser.
     password);
  if(!isCorrectHash) {
    let error = new Error('Provided Login credentials are wrong');
    error.status = 401;
    return done(null, false);
  }

  //login success
  return done(null, dbUser);
})
);

/**
*  Configure Passport authenticated session persistence.
*/
passport.serializeUser(function(ethereumAddress, done) {
  done(null, ethereumAddress);
});

passport.deserializeUser(function(user, done) {
  done(null, user);
});

export default passport;
```

Figure 4.46: local-passport.mjs: Own defined Passport Strategy

- **Email**
  These modules are responsible for managing the email forwarding. 'index.js' has the function 'sendEmailByEthereumAddress()' which is capable to send various email templates provided as parameter to a User given its ethereum address. It gets the email address from the data base and sends the personalized email using user data. The subfolder further contains the different templates, which are used for send different mail for various scenarios. To mention is that not all email templates necessary are created, as we did not consider it highly necessary for the Minimum viable product (MVP).

- **Web3**
  These modules handle all direct interaction of the back-end application with the SC. 'Contract-event-listener.mjs' is responsible for listen to all events happening in the SC. 'Timebased-contract-service.mjs' triggers function on the SC based on time exceeding. Their functionalities and interaction is explained in detail in Subsections 4.7.3 and 4.7.4

### 4.5.5 REST interface

This subsection shows the REST-interface used for the MVP of the convenience service. We present the most important routes of the communication paths of the back-end, leaving out routes which are not considered crucial for explaining the token interaction.

Table 4.1 shows the whole REST interface considering the user management. General information about users can be retrieved with the call on the first line. Personal information and more detailed information can be retrieved with the second and third call. The backend application always takes the Ethereum address provided by the token as a filter for the response data, this assures that a user only can request his/her personal detailed data. Further information about personal roles as well as all historical transactions performed on the SC can be retrieved.
Interaction with article-drafts are organized on the endpoints listed in table 4.2. A user can get all its existing article-drafts or get a specific by providing the id as a path parameter. Before submitting or reverting an article-draft in the SC, the users-platform makes use of one of the two last requests of the table to notify the back-end accordingly.
An article-draft once submitted in the SC triggers the second request in the table 4.2, which then creates a article-submission and article-version for the user accordingly. Endpoints for article-versions are listed in table 4.3. A user can request all of its article-versions. He can use multiple filters on them by providing some query parameters. When a user is assigned as editor to an article-version, he can sign him off or invite reviewers with request four and five.
The interaction with article-submission is minimal as shown in table 4.4. A user can retrieve all of its own article-submission. Or it can filter them by assigned or unassigned article-submissions.
All request considering reviews are listed in table 4.5. Reviews can be retrieved isolated by their different state. Request one to three returning your reviews filtered by different states. The forth request is used to invite others for reviewing. Whereas request five, six and seven update and create a review.

Table 4.1: REST interface: Users

| URL | Method | Description | Authorized Role | Query parameters |
|---|---|---|---|---|
| **/users** 'ADMIN' | GET ethereumAddress: String | Get a user by the Ethereum address | 'CONTRACT_OWNER' | |
| **/users/roles** | GET | Get all users by roles | all | role: Role |
| **/users/data** | GET | Get all detail infos about the user | all | ethereumAddress: String |
| **/users/ownRoles** | GET | Get all roles of itself | all | ethereumAddress: String |
| **/users/ownRoles** | PUT | Add roles to itself | all | role: Role (only 'AUTHOR' or 'REVIEWER') |
| **/users/ownRoles** | DELETE | Delete roles from itself | all | role: Role (only 'AUTHOR' or 'REVIEWER') |
| **/users/scTransactions** | GET | Get transactions from itself | all | ethereumAddress: String |

Table 4.2: REST interface: Article-Drafts

| URL | Method | Description | Authorized Role | Query parameters |
|---|---|---|---|---|
| **/articleDraft** | GET | Get all the article-drafts from yourself | all | ethereumAddress: String |
| **/articleDraft/new** | GET | Create a new articleSubmission with a first articleVersion in it | 'AUTHOR' | ethereumAddress: String |
| **/articleDraft/:draftId** | GET | Get a specific article-version, but only if it is a draft | all | ethereumAddress: String |
| **/articleDraft/:draftId** | PUT | Update the article-draft | 'AUTHOR' | ethereumAddress: String, document: Object{} |
| **/articleDraft/:draftId/submit** | PUT | Remark the backend, that the user will submit this article-draft. (Backend will match a submission with the articleHash) | 'AUTHOR' | ethereumAddress: String, articleHash: String |
| **/articleDraft/:draftId/revert** | PUT | Revert the article-Draft to state 'DRAFT', when it has not been submitted on the SC yet. | 'AUTHOR' | etherumAddress: String, |

Table 4.3: REST interface: Article-Versions

| URL | Method | Description | Authorized Role | Query parameters |
|---|---|---|---|---|
| **/articleVersions** | GET | Get all the article-version matching the query parameters | all | ethereumAddress: String, title: String, page: int, limit: int |
| **/articleVersions/:id** | GET | Get a specific article-version by its id | all | ethereumAddress: String id: String |
| **/articleVersions/submitted** | GET | Get submitted and finished article-versions of the User | all | ethereumAddress: String |
| **/articleVersions/assigned/signoff** | GET | Get all submitted article-versions of the User | 'EDITOR' | ethereumAddress: String |
| **/articleVersions/assigned/inviteReviewers** | GET | Get all article-versions with state 'OPEN_FOR_ALL_REVIEWERS' | 'EDITOR' | ethereumAddress: String |
| **/articleVersions/assigned/finalize** | GET | Get all article-versions which are ready to finalize | 'EDITOR' | ethereumAddress: String |
| **/articleVersions/reviewable** | GET | Get all your articles-version open for review | 'REVIEWER' | ethereumAddress: String |
| **/articleVersions/reviewable/expert** | GET | Get all your article-version open for invited reviewers | 'REVIEWER' | ethereumAddress: String |

Table 4.4: REST interface: Article-Submissions

| URL | Method | Description | Authorized Role | Query parameters |
|---|---|---|---|---|
| **/articleSubmission** | GET | Get all of your articleSubmissions | all | ethereumAddress: String |
| **/articleSubmission/unassigned** | GET | Get all unassigned ArticleSubmission ('OPEN' && no editor) | all | ethereumAddress: String |
| **/articleSubmission/assigned** | GET | Get all assigned ArticleSubmission ('CLOSED' ) | all | ethereumAddress: String |

Table 4.5: REST interface: Reviews

| URL | Method | Description | Authorized Role | Query parameters |
|---|---|---|---|---|
| **/reviews/invited** | GET | Get all of your reviews which are in state 'INVITED' or 'SIGNED_UP_FOR_REVIEWING' | all | ethereumAddress: String |
| **/reviews/myreviews** | GET | Get all your reviews | all | ethereumAddress: String |
| **/reviews/checkable** | GET | Get all of your reviews which are in state "HANDED_IN_SC" | all | ethereumAddress: String |
| **/reviews/:reviewId** | GET | Get review by the reviewId | all | |
| **/reviews/invite** | POST | Create a Review Invitation (also creates a new Review, if one with the articleHash does not exist yet) | 'AUTHOR' | reviewerAddress: String, articleHash: String, reviewType: ReviewType |
| **/reviews/editorApproved** | PUT | Add an review incl. scores and issues into an existing review | 'REVIEWER' | ethereumAddress: String, reviewId: String, reviewText: String, reviewHash: String, score1: int, score2: int, articleHasMajorIssues: bool, articleHashMinorIssues: bool |
| **/reviews/update** | PUT | Update an existing review with new content | 'REVIEWER' | ethereumAddress: String, reviewId: String, reviewText: String, reviewHash: String, score1: int, score2: int, articleHasMajorIssues: bool, articleHashMinorIssues: bool |
| **reviews/community** | PUT | Add a new community review for a specific article | all | ethereumAddress: String, reviewId: String, reviewText: String, reviewHash: String, score1: int, score2: int, articleHasMajorIssues: bool, articleHashMinorIssues: bool |

## 4.6 Interface between front-end and smart contracts

### 4.6.1 React, Web3 and MetaMask

For the integration with the Ethereum blockchain, we used Web3 [6], a library collection that allows you to interact with a local or remote ethereum node through a HTTP or IPC connection. Concerning Ethereum key-pairs management we rely on MetaMask [15], which acts as a bridge in your Browser for performing Ethereum transactions. The latter and Web3 are highly coupled, where Web3 provides the library, methods and APIs and MetaMask acts as a browser-based Provider exposing the Ethereum accounts. The Figure 4.5 provides a more detailed architecture about the inter-dependencies between MetaMask and Web3 in the React ecosystem. The initialization process of Web3, the interactions with the Ethereum Blockchain and their inter-connections have been discussed in the Section 4.3.1.

### 4.6.2 Web3 contract methods interface

Section 4.4.4 explains the smart contract interface which can be used by the front-end to call the smart contract methods without any special web3 dependent syntax.

### 4.6.3 Sending transactions and resolving promiEvents

In a user friendly front-end every action can be called from not only one specific point. The goal is always that the user can perform an action from different starting points. Because of that we created a folder called ArticleActions in webpack/components/Articles. It includes a file for every action which can be performed on the article or the article submission process. This includes *EditArticle*, *SignUpForExpertReview*, *DeclineArticleAndClose* and so on. For example SignUpForExpertReview can either be called from the view, where all possible articles open to review are listed or from one detailed article view.

One of these files usually includes the action itself (so the methods to call the smart contract and wait for the response) and the button which can be placed at different places in the front-end.

Listing 4.18 shows the AssignAsEditorButton which returns the ActionButton object. In line 13-17 one can see, when the Button is clicked the assignEditor function is called and when it returns with a callback that the article is update by fetchingArticleData again. The button itself takes the `ASSIGN_ARTICLE` label and tooltip.

This assignEditor method which is called by clicking the button is shown in listing 4.19. Every function communicating with the smart contract does a gas estimation first (line 3-12). For developing with Ganache the gasAmount needs to be fixed very high. After the gas estimation returned the same method from the web3 method interface is called again, instead of `.estimateGas()` to send the transaction to the Ethereum blockchain

the `.send()` function is used.

`.send()` is a web3.js function and returns a promiEvent [7]. In the EUREKA interface every `.send()` function is followed by two event handlers and one error handler. When the Ethereum node return the transaction hash (line 25-32, `.on('transactionHash')`) the front-end adds this transaction hash to the transaction pool and toasts a first notification that the transaction is pending and being processed in the next minutes.

When the send-function returns the second event, it is hopefully the transaction receipt (line 35-44, `.on('receipt')`) which means that the transaction has been successfully processed. On this receipt the front-end toast the second notification which informs the user that th article has been successfully assigned to the user. It also calls the callback function which, as seen before, updates the article data.

If the send-function receives an error, then it is catched by the catch-clause and an error notification is displayed (line 46-48, `.catch()`).

```
export const AssignAsEditorButton = connect(
  mapStateToProps,
  mapDispatchToProps
)(props => {
  return (
    <Web3Context.Consumer>
      {web3Context => {
        return (
          <ActionButton
            dataTip={'assignEditor'}
            icon={'editorAssign'}
            background={__FIFTH}
            onClick={() => {
                assignEditor(web3Context, props, () => {
                  props.fetchingArticleData(props.article._id);
                });
            }}
            title={ASSIGN_ARTICLE.tooltip}
          >
            {ASSIGN_ARTICLE.label}
          </ActionButton>
        );
      }}
    </Web3Context.Consumer>
  );
});
```

Listing 4.18: The AssignAsEditorButton, an example for all smart contract functions which can be called from the user by clicking a button

```
export const assignEditor = async (web3Context, props) => {

  let gasAmount;
  // gas estimation on ganache doesn't work properly
  if (!isGanache(web3Context.web3))
    gasAmount = await assignForSubmissionProcess(
      web3Context.platformContract,
      props.article.articleSubmission.scSubmissionID
```

```
9       ).estimateGas({
10        from: props.selectedAccount.address
11      });
12    else gasAmount = 80000000;
13
14
15    assignForSubmissionProcess(
16      web3Context.platformContract,
17      props.article.articleSubmission.scSubmissionID
18    )
19
20    .send({
21      from: props.selectedAccount.address,
22      gas: gasAmount
23    })
24
25    .on('transactionHash', tx => {
26      props.addTransaction(SC_TRANSACTIONS_TYPE.EDITOR_ARTICLE_ASSIGNMENT,
            tx);
27      toast.info(
28        <EditorInfoMessage
29          path={'signoff'}
30          text={'Your article will be assigned to you in the next minutes.
              '}
31        />
32      );
33    })
34
35    .on('receipt', receipt => {
36      toast.success(
37        <EditorSuccessMessage
38          path={'signoff'}
39          articleId={props.article._id}
40          text={`The article has been successfully assigned to yourself`}
41        />
42      );
43      callback();
44    })
45
46    .catch(err => {
47      toast.error(err.toLocaleString(), {autoClose: false});
48    });
49  };
```

Listing 4.19: The assignEditor function which is called when the AssignAsEditorButton is clicked.

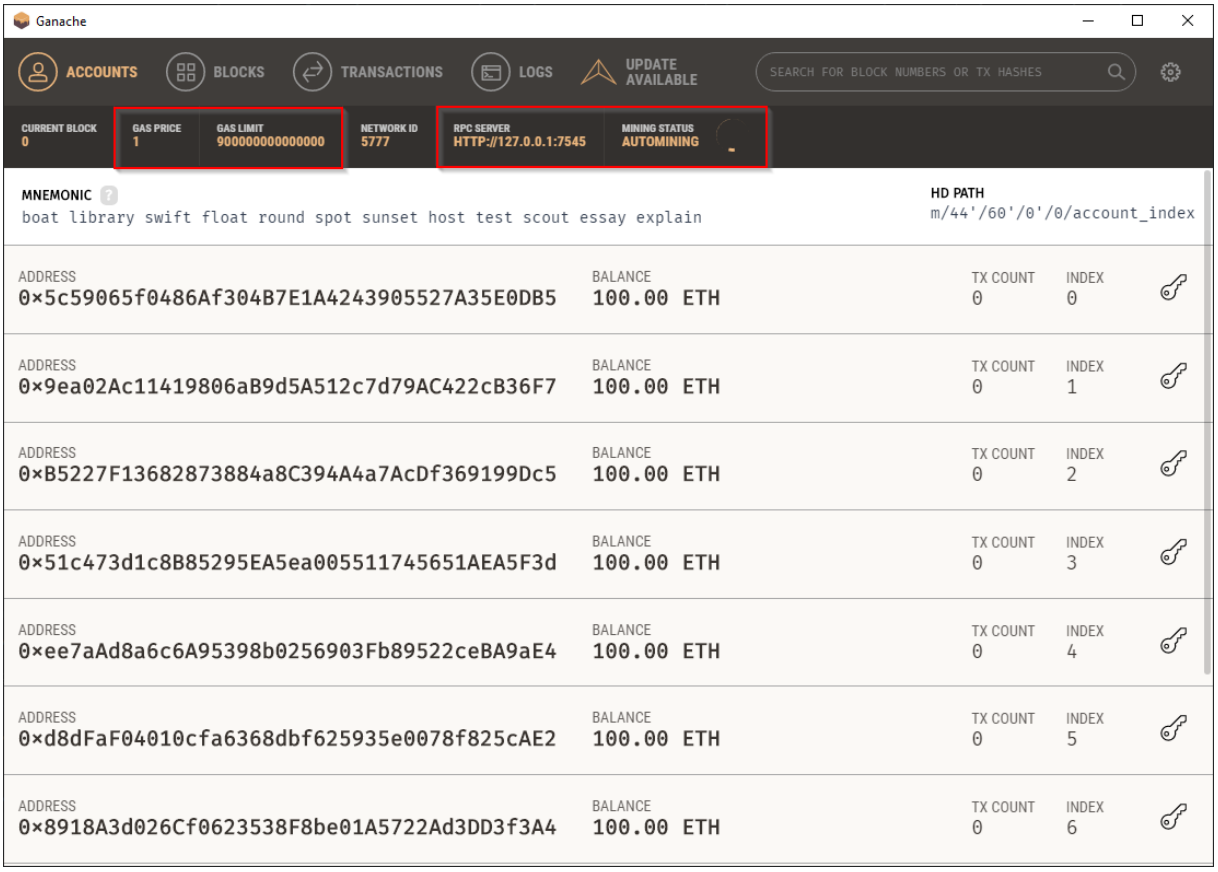



Figure 4.47: The local Ethereum blockchain provider Ganache

## 4.7 Interface between smart contracts and back-end

### 4.7.1 Smart contract deployment

The EUREKA source code also includes a smart contract deployment script. Currently it is only used for the development environment.

#### Development environment

In the development environment the smart contracts are deployed to the ”one click blockchain” Ganache [16] by the truffle framework. It is providing an Ethereum-like blockchain locally. Users can specify the gas price, gas limit and the mining block time (see red areas in Figure 4.47). The mining block time can be kept very low (e.g. ”automine” after every transaction) for not having delays during development.

Ganache is not persistent. When it is closed or the machine is shut down, the deployed smart contracts are lost.

This leads to the need of fast and easy smart contract deployment. The EUREKA source

code includes a deployment script which can be used for automated deployment. With `npm run deploy-contracts` the `BC_NETWORK` and the `NODE_ENV` environment variables are set to 'ganache' respectively 'development' and the 'deployer-script' in smartcontracts/deployment is run.

In smartcontracts/deployment/get-input.mjs the name and locations of the smart contracts and the libraries can be specified. The deployer-script works sequentially: First, all libraries are deployed and in a second step the smart contracts do follow. This allows the use of import-statements in the Solidity smart contracts.

Listing 4.20 shows that the deployer-script not only deploys the contracts, it also mints the EKA tokens and saves the token contract address to the platform contract (line 10-11) automatically.

```
1  const run = async () => {
2
3    await deployAndMint();
4    process.exit();
5  };
6
7  const deployAndMint = async () => {
8
9    const [eurekaTokenContract, eurekaPlatformContract] = await
       deployContracts();
10   await mintEKATokens(eurekaTokenContract);
11   await setEurekaTokenAddress(eurekaPlatformContract,
       eurekaTokenContract.options.address);
12 }
13
14 run();
```

Listing 4.20: The high-level code of the deployer-script, implemented by the developers of EUREKA

### Test environment

The smart contracts have been deployed to the Kovan test network [67] by using the Remix IDE of the Ethereum organisation [68]. The smart contracts can very easily be compiled and deployed to different Ethereum networks using MetaMask [15] (see figure 4.48).

### Production environment

The smart contracts have not been deployed to the main network yet.

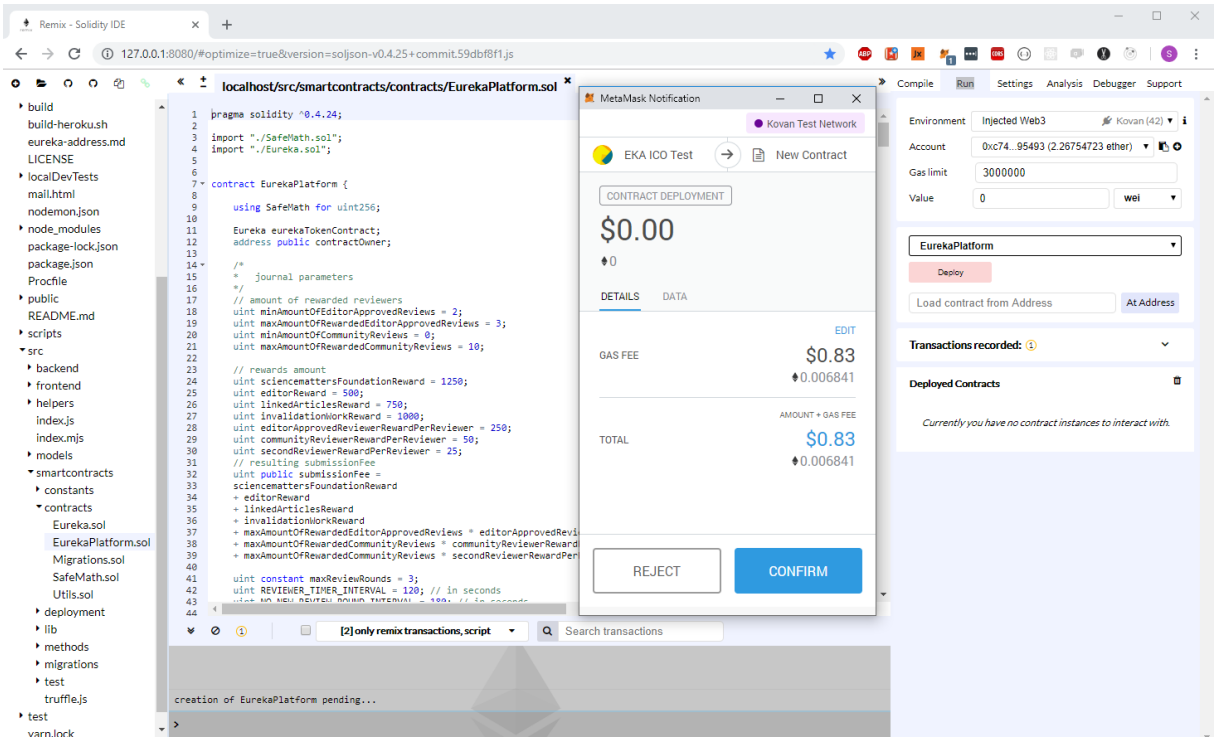



Figure 4.48: The Ethereum Solidity IDE Remix after clicking ”Deploy” and before confirming the transaction with MetaMask

### 4.7.2 Connection to Ethereum node

To communicate with the Ethereum blockchain for the above mentioned services (4.7.3), the EUREKA back-end needs to connect to an Ethereum node. This Ethereum node is different for every environment.

When the back-end server is booted, the web3 interface need to be setup. For this purpose the setupWeb3Interface method in backend/web3/web3InterfaceSetup.mjs is called.

As one can see in listing 4.21 (line 9, 15 and 21), the server connects to the blockchain network which is specified in the environment variable `BC_NETWORK`. The method reads different contract addresses and ABIs depending on what blockchain network is specified (line 10-13, 16-19 or 22-25). In line 39 and 40 the contract objects are created: the web3 instance is used to that. When platformContract and the tokenContract objects are created, the contractEventListener is set up (line 47) and the journal information which are pulled from the smart contract are saved to the database.

```
1 import web3 from '../../helpers/web3Instance.mjs';
2 ...
3 export const setupWeb3Interface = async (_platformContract,
    _tokenContract) => {
4
5   let platformContractAddress;
6   let platformContractABI;
7   let tokenContractAddress;
8   let tokenContractABI;
```

```
9   if (process.env.BC_NETWORK === 'main') {
10    platformContractAddress = PLATFORM_MAIN_ADDRESS;
11    platformContractABI = PLATFORM_MAIN_ABI;
12    tokenContractAddress = TOKEN_MAIN_ADDRESS;
13    tokenContractABI = TOKEN_MAIN_ABI;
14  }
15  else if (process.env.BC_NETWORK === 'ganache') {
16    platformContractAddress = GanachePlatformContractAddress;
17    platformContractABI = GanachePlatformContractABI;
18    tokenContractAddress = GanacheTokenContractAddress;
19    tokenContractABI = GanacheTokenContractABI;
20  }
21  else if (process.env.BC_NETWORK === 'kovan') {
22    platformContractAddress = PLATFORM_KOVAN_ADDRESS;
23    platformContractABI = PLATFORM_KOVAN_ABI;
24    tokenContractAddress = TOKEN_KOVAN_ADDRESS;
25    tokenContractABI = TOKEN_KOVAN_ABI;
26  }
27  else {
28    console.error('BC_NETWORK ' + process.env.BC_NETWORK + ' couldn\'t
        be found');
29    process.exit(1);
30  }
31
32
33  let platformContract;
34  let tokenContract;
35
36  // if server is started up without contract deployment (if contracts
       already exists)
37  if (typeof _platformContract === 'undefined'
38  && typeof _tokenContract === 'undefined') {
39    platformContract = new web3.eth.Contract(platformContractABI,
        platformContractAddress);
40    tokenContract = new web3.eth.Contract(tokenContractABI,
        tokenContractAddress);
41  }
42  else {
43    platformContract = _platformContract;
44    tokenContract = _tokenContract;
45  }
46
47  await contractEventListener.setup(platformContract);
48  await saveJournalInformation(platformContract);
49
50  return [platformContract, tokenContract];
51 };
```

Listing 4.21: The setupWeb3Interface which creates the smart contract objects of the specified blockchain network

The first line of listing 4.21 indicates the initialization of the web3 instance. This web3 instance is used to create the smart contract objects afterwards.

When importing the web3Instance the web3 provider is initialized (see listing 4.22). The environment variable `BC_NETWORK` is used to distinguish which provider respectively

Ethereum node should be connected.

In the following sections, the different Ethereum node setups are described.

```
1  import Web3 from 'web3';
2  const web3 = new Web3();
3
4  const initProvider = () => {
5    web3.setProvider(getProvider());
6  };
7
8  const getProvider = () => {
9    let provider;
10   if (process.env.BC_NETWORK === 'main') {
11     provider = new Web3.providers.WebsocketProvider('wss://infura.io/ws'
         );  // in future connecting to digital ocean ethereum-node: 'ws
         ://' + process.env.ETHEREUM_FULL_NODE
12   }
13   else if (process.env.BC_NETWORK === 'ganache') {
14     provider = new Web3.providers.WebsocketProvider('ws://127.0.0.1:7545
         ');
15   }
16   else if (process.env.BC_NETWORK === 'kovan') {
17     provider = new Web3.providers.WebsocketProvider('ws://' + process.
         env.ETHEREUM_FULL_NODE_KOVAN);   //connecting to digital ocean
         ethereum-kovan-node
18   }
19   else {
20     console.error('provider ' + process.env.BC_NETWORK + ' couldn\'t be
         found');
21     process.exit(1);
22   }
23   ...
24   return provider;
25 };
26
27 initProvider();
28 export default web3;
```

Listing 4.22: The web3Intance.mjs which creates the smart contract objects of the specified blockchain network

### Main network for production

If the `BC_NETWORK` variable is set to 'main' the server connects to the main Ethereum network. At the moment it is set to the provider Infura [17] but in the future it will be connected to EUREKA's Ethereum full node running on a Digital Ocean droplet [66].

### Kovan network for test environment

When working in the test environment but also sometimes in development, the `BC_NETWORK` variable shall be set to 'kovan'. The web3 instance will then be initiated with the websocket

provider connecting to EUREKA's Ehtereum Kovan network full node. It is running on a Digital Ocean droplet too.

**Ganache for development environment**

For connecting to Ganache the `BC_NETWORK` variable needs to be set to 'ganache'. The Ganache port number can be defined in Ganache as well. As one can see the port number from the WebsocketProvider in line 14 of listing 4.22 is the same as in figure 4.47 surround by a red border (7545).

### 4.7.3 Smart contract mirror - contract event listener

All action happening on the SC need to be tracked by the back-end application in order to keep it up to date with all changing states. For that we implemented a so called **SC Monitor**.
The file 'Contract-event-listener.mjs' is responsible for monitoring all events defined in the SC. The back-end application registers itself for all events of the SC in order to mirror actions executed within it. Figure 4.49 shows one event listener example, which handles the event, when a new editor gets assigned to an article-submission. It calls the function 'updateEditorToSubmission' from the 'articleSubmissionService', passing the ID of the article-submission and the ethereum address of the new editor as parameter, provided by the event of the SC ( lines 8 - 11). This functions returns the ethereum address of the old editor, which on the other hand is used to notify the old editor by mail, that he has been removed as editor from the article-submission. This email is sent by implementing the email-service mentioned above (line 14 - 19).

```
1  ...
2  /** Change editor from submission process **/
3  EurekaPlatformContract.events.ChangedEditorFromSubmission(
4  undefined,
5  async (error, event) => {
6  if (error) throw error;
7
8  const removedEditorAddress = await articleSubmissionService.
       updateEditorToSubmission(
9  event.returnValues.submissionId,
10 event.returnValues.newEditor
11 );
12
13 // Email notification to removed Editor
14 sendEmailByEthereumAddress({
15 ethereumAddress: removedEditorAddress,
16 from: 'info@eurekatoken.io',
17 subject: 'Editor Resignation',
18 html: getEditorResignedInvitationTemplate()
19 });
20 }
21 ...
```

Figure 4.49: contract-event-listener.mjs: Handling all events from SC in the backend

### 4.7.4 Timeout handler - time-based contract service

The 'time-based-contract-service.mjs' is responsible for triggering functions in the smart-contract based on time-constraints. The idea is that various states of an article-submission, which are not changed by a user within a given time-frame, will be changed by this module, as otherwise submissions would get stuck and no one can proceed with it. Figure 4.50 shows the implementation of the case that editor gets removed from article-submission if they do not proceed with the sanity check within the given time-frame.
The implemented cron-job (line 7-10) calls the function 'getEditorTimeoutSubmissionIds' (line 9) from Figure 4.51 on a regularly basis.This functions returns all articleSubmission IDs from the database where the editors did not performed a Sanity check within its given time frame.These IDs get then passed into the function 'removeEditorFromSubmissionbyScSubmissionId' (line 10). In Figure 4.52 it calls for each ID the SC-function 'removeEditorFromSubmissionProcess' providing the ID as paramter (line 7-12).

```
export default {
start: async (_platformContract, _contractOwnerAddress) => {
platformContract = _platformContract;
contractOwnerAddress = _contractOwnerAddress;


cronJob = await new CronJob(CRONE_TIME_INTERVAL, async () => {
// Remove editor which don't do sanity check
const timedOutSubmissionIds = await getEditorTimeoutSubmissionIds();
await removeEditorFromSubmissionbyScSubmissionId(timedOutSubmissionIds
    );
// TODO inform editor by mail and in frontend
```

Figure 4.50: timebased-contract-service.mjs: Handling time-based triggers of SC-functions

```
1
2   /**
3   * First get all the submission ids from the article-version,
4   * where the state is 'SUBMITTED' and the articleVersion State is '
        EDITOR_ASSIGNED'.
5   * Secondly, it checks if the time for the Editor to do the health-
        check is ran out.
6   * If that is the case the submissionId of the corresponding
        Articlesubmission gets saved
7   * and eventually returned with all other submissionIds.
8   */
9   async function getEditorTimeoutSubmissionIds() {
10    const submittedArticleVersions = await articleVersionService.
          getArticleVersionsByState(ArticleVersionState.SUBMITTED);
11    let potentialTimedOutVersions = [];
12
13    for(let submittedArticleVersion of submittedArticleVersions) {
14      const correspondingArticleSubmission =
15      await articleSubmissionService.getSubmissionById(
            submittedArticleVersion.articleSubmission);
16      if(correspondingArticleSubmission.articleSubmissionState ===
            ArticleSubmissionState.EDITOR_ASSIGNED) {
17        potentialTimedOutVersions.push({articleVersion:
              submittedArticleVersion, submissionTimestamp:
              correspondingArticleSubmission.stateTimestamp, scSubmissionID
              : correspondingArticleSubmission.scSubmissionID});
18      }
19    }
20
21    let timeoutSubmissionIds = [];
22
23    let differences = [];
24    let scIDs = [];
25
26    for (let potentialTimedOutVersion of potentialTimedOutVersions) {
27      const now = Math.round(new Date().getTime()/1000);
28      const timestamp = potentialTimedOutVersion.submissionTimestamp;
29
30      // only testing TODO remove
31      scIDs.push(potentialTimedOutVersion.scSubmissionID);
32      differences.push(now - timestamp);
33
34      if((now - SANITY_TIME_OUT_INTERVAL - timestamp) > 0) {
35        timeoutSubmissionIds.push(potentialTimedOutVersion.
              scSubmissionID);
36      }
37    }
38    return timeoutSubmissionIds;
39  }
```

Figure 4.51: timebased-contract-service.mjs: Get IDs of Editor with a time-out

```
1
2   async function removeEditorFromSubmissionbyScSubmissionId(
       timedOutSubmissionIds) {
3     if(timedOutSubmissionIds.length > 0) {
4
5       for(let timedOutSubmissionId of timedOutSubmissionIds) {
6         try{
7           await removeEditorFromSubmissionProcess(
8           platformContract,
9           timedOutSubmissionId
10          ).send({
11            from: contractOwnerAddress
12          });
13        } catch (e) {
14        console.warn('Removing of Editor is already done, ERROR-MESSAGE:
              ' + e);
15      }
16    }
17  }
18  return 'Remove Editors done';
19  }
```

Figure 4.52: timebased-contract-service.mjs: Remove Editors from Submission on the SC

## 4.8 Interaction between all components

The interaction between all components is described by looking at one transaction in detail.

Figure 4.53 shows the detailed process which the platform is going through when it interacts with the Ethereum blockchain. It includes the subprocesses *inviting a reviewer*, *signing up for reviewing an article*, *writing a review* and the first part of *submitting a review*. It looks at the roles handling editor and expert reviewer, at the service EUREKA front-end, the back-end services email service, smart contract monitor and database and at the blockchain services Ethereum node and the EUREKA platform contract.

When the handling editor decides to invite an expert reviewer that he is informed about a new article to review and sees the article as soon as possible, the editor can enter the email addresses in the front-end and enter e personal message if he wants to. The email is then sent by the email service of the back-end.

The expert reviewer can then follow a link to the platform in the email. There he can decide if he wants to sign up for a review or if he wants to draft the review first. Signing up for a review will result in a reserved reward for a defined (from the journal owner) amount of time (e.g. three days). By clicking the ”signing up” button the Ethereum interface is prompted and the reviewer has to sign the transaction and confirm the amount of gas which needs to be paid for this transaction. If that happened, the front-end sends the signed transaction to the blockchain. The Ethereum will respond almost immediately with a transaction hash. The front-end shows a notification that the transaction has been

successfully sent, hand the transaction hash is saved to the database that the user can track his sent transaction all the time and on every device.

As soon as the transaction is confirmed two different things happen: On the front-end side the *promiEvent* [7] returns a transaction receipt. But this is only used to trigger the second notification for the user and to update the transaction status on the database. No other database changes triggered by this receipt. The platform contract emits an event as soon as the changes were successful. Since the monitor is listening on this event the state changes are mirrored to back-end respectively the database.

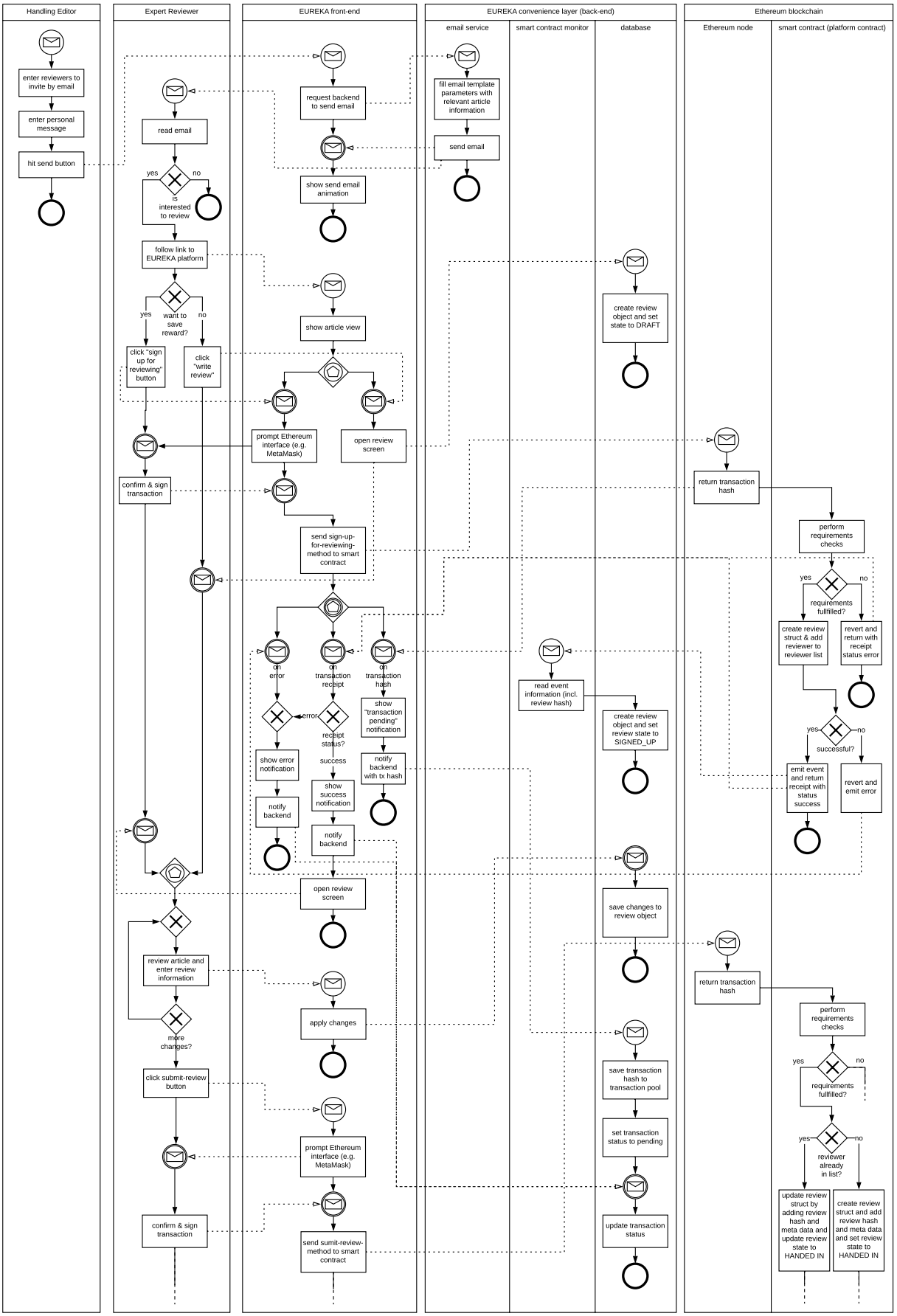



Figure 4.53: The sign-up-for-reviewing transaction in detail

# Chapter 5

# Evaluation

To assure a good quality of product in technical terms as well as in usability, we included two main provision presented in the following chapter. First the integration tests developed for the system source code is presented in detail. Secondly, the System usability scale applied to our platform is presented with its result concerning the usability of our platform.

## 5.1 Integration Tests

In order to test the proper behavior and interplay between the different systems of the SC and back-end application we make the use of integration tests. The idea of integration test is to combine these two individual units and test them as a group. The goal is to expose faults in the interaction between the SC and back-end application. We wrote these integration tests parallel to developing the platform and they actually help us a lot as they exposed various bugs and illogical development, which we could then fix and redesign in a early stage.

The integration test are all build in the pattern described in the following:

1. Setup up the data and environment simulating the situation to test.
2. Call a function which triggers the corresponding Event on the SC.
3. Through the event-monitor on the back-end side the corresponding function gets called with parameters coming from the SC event.
4. Automated manipulation on the database are made accordingly.
5. The test function checks if the values within the database are changed as expected

With this test pattern we can assure that action done in the SC are correctly traced by the monitor and the platform data behaves as expected. Therefore we provide correct data to the front-end and the end-user. In Listing **??** a example integration-test for assigning a user to become an editor is shown.
It first saves the number of roles and also the number of scTransactions the user has before he gets signed up (line 19). A function gets called, which informs the SC to sign up the passed ethereum address to become an editor (lines 24). As there is no possibility to tell the compiler now to wait for all changes are completely written into the database, we came up with the following work-around approach: We check for multiple times (5 in example code) if the number of scTransactions has already increased. If this is not the case, the compiler sleeps for a while and start the same loop again (lines 32-39). After a maximum of 5 times the loop finishes and the data from the database is tested. In the example we check if the user has now a new added role, which has to be 'EDITOR' (line 40-41).

Some integration tests, like the one introduced above, are responsible to prove that the interchange of one function of the SC is working properly. To be sure that the whole business logic works as expected, we also need to simulate a whole process from beginning to end. We designed all the single integration tests in a way that they can get chained together, all manipulating and testing for their own. Most of the integration test are multiple, cascading request on the SC. The goal of these is to test a whole process case from beginning to end.
Figure 5.2 and reffig:backend-integration-test-all-2 show an integration test example, which goes through the whole process of author creation up to article submission. It combines multiple single integration test (e.g. the signUpEditor form above on line 64). If only a single assertion in one of all these test is failing, the overall integration test will fail.

## 5.2 System Usability Scale (SUS)

System Usability Scale [69] is a user-oriented testing strategy for measuring the usability of a certain system. With this testing approach, participants are asked to score different questions with one of five responses that range from Strongly Agree to Strongly disagree. The questionnaire and the corresponding scoring system are outlined in the System Usability Scale (SUS) Template [70]. Due to the fact that the EUREKA platform can still be considered a minimum viable product for early customers, we decided to perform this testing approach because it can be used on small sample sizes and obtaining reliable results. Furthermore, it is a very easy scale to administer to participants. It is worth to point out, that the usability of any software has to be considered in terms of the context in which it is used and therefore its appropriateness has to be viewed in that context [70]. In order to obtain a reliable SUS measurement for the EUREKA platform, we decided to interview twelve people having different background and skill sets. We defined and divided our participants into three different groups based on the following criteria:

- *First Group - Editorial Team of EKA Blockchain Solutions GmbH*
  The first group is composed by three people working in the Editorial Team of the

```
/************************ Sign up Editor ************************/
test(PRETEXT + 'Sign up Editor', async t => {
let user = await createUserContractOwner();
t.is(user.roles.length, 1);
t.is(user.roles[0], Roles.CONTRACT_OWNER);

await TestFunctions.signUpEditorAndTest(t, user);
});


/**
*  Signs up the provided user as an Editor on the SC.
*  Afterwards checks if the user has become an Editor on the DB as well
* @param t
* @param user
* @returns {Promise<*>}
*/
signUpEditorAndTest: async function(t, user) {
const rolesLength = user.roles.length;
const scTransactionLength = user.scTransactions.length;
let dbUser = await userService.getUserByEthereumAddress(user.
    ethereumAddress);

t.is(dbUser.roles.includes(Roles.EDITOR), false);
await signUpEditor(eurekaPlatformContract, user.ethereumAddress).send({
from: contractOwner
});

dbUser = await userService.getUserByEthereumAddressWithScTransactions(
user.ethereumAddress
);

let counter = 0;
while (dbUser.scTransactions.length < scTransactionLength && counter <
    5) {
sleepSync(5000);
dbUser = await userService.getUserByEthereumAddressWithScTransactions(
contractOwner
);
counter++;
}
t.is(dbUser.roles.length, rolesLength + 1);
t.is(dbUser.roles[rolesLength], Roles.EDITOR);
return t;
},
```

Figure 5.1: One of many integration tests, testing the proper interaction between smart-contract and back-end when an user gets editor-assigned
, code:backend-integration-test

```
/*************** Article acception *****************/
test(
PRETEXT +
'Article acception',
async t => {
// Create author and editor
const author = await createUserContractOwner();
const editor = await createEditor1();
const reviewer1 = await createReviewer1();
const reviewer2 = await createReviewer2();
const reviewer3 = await createReviewer3();
const editorApprovedReviewers = [reviewer1, reviewer2, reviewer3];
const reviewer4 = await createReviewer4();

// Setup article draft
await TestFunctions.createArticleDraftAndSubmitIt(t, author,
    TEST_ARTICLE_1_HASH_HEX, TEST_ARTICLE_1_DATA_IN_HEX);
let articleSubmission = (await articleSubmissionService.
    getAllSubmissions())[0];
let articleVersion = articleSubmission.articleVersions[0];

// Signup editor & assign him on article
await TestFunctions.signUpEditorAndTest(t, editor);
await TestFunctions.assignEditorForSubmissionProcess(t, editor,
    articleSubmission);

// Accept sanity check for article 1
await TestFunctions.acceptSanityCheckAndTest(t, editor, author,
    articleVersion);

// Invite reviewers 1,2 & 3 to become editor-approved reviewers
await TestFunctions.inviteReviewersAndTest(t, editor, author,
    editorApprovedReviewers, articleVersion);

//update from DB
articleSubmission = (await articleSubmissionService.getAllSubmissions())
    [0];
articleVersion = articleSubmission.articleVersions[0];

await TestFunctions.acceptInvitationAndTest(t, editorApprovedReviewers
    [0], 0, articleVersion);
await TestFunctions.acceptInvitationAndTest(t, editorApprovedReviewers
    [1], 1, articleVersion);
await TestFunctions.acceptInvitationAndTest(t, editorApprovedReviewers
    [2], 2, articleVersion);
```

Figure 5.2: Integration test 1/2: It tests the whole process from author creation up to an article submission

```
37 // Get the reviews from DB
38 let review1 = await reviewService.getReviewById(reviewer1.
       ethereumAddress, articleVersion.editorApprovedReviews[0]);
39 let review2 = await reviewService.getReviewById(reviewer2.
       ethereumAddress, articleVersion.editorApprovedReviews[1]);
40 let review3 = await reviewService.getReviewById(reviewer3.
       ethereumAddress, articleVersion.editorApprovedReviews[2]);
41
42 // Add editor-approved review into DB
43 await TestFunctions.addEditorApprovedReviewAndTest(t, reviewer1, review1
       , NO_ISSUES_REVIEW_1, NO_ISSUES_REVIEW_1_HASH_HEX, articleVersion);
44 await TestFunctions.addEditorApprovedReviewAndTest(t, reviewer2, review2
       , NO_ISSUES_REVIEW_2, NO_ISSUES_REVIEW_2_HASH_HEX, articleVersion);
45 await TestFunctions.addEditorApprovedReviewAndTest(t, reviewer3, review3
       , MINOR_ISSUES_REVIEW_1, MINOR_ISSUES_REVIEW_1_HASH_HEX,
       articleVersion);
46
47 //update from DB
48 articleSubmission = (await articleSubmissionService.getAllSubmissions())
       [0];
49 articleVersion = articleSubmission.articleVersions[0];
50
51 // Add a community review as reviewer4
52 await TestFunctions.addNewCommunitydReviewAndTest(t, reviewer4,
       MINOR_ISSUES_REVIEW_2, MINOR_ISSUES_REVIEW_2_HASH_HEX, author,
       articleVersion);
53
54 // Accept review1 & review2, decline review3
55 await TestFunctions.acceptReviewAndTest(t, editor, review1,
       articleVersion);
56 await TestFunctions.acceptReviewAndTest(t, editor, review2,
       articleVersion);
57 await TestFunctions.declineReviewAndTest(t, editor, review3,
       articleVersion);
58
59 // Accept ArticleVersion
60 await TestFunctions.acceptArticleVersionAndTest(t, editor,
       articleVersion);
61
62 // Test if articleSubmission is closed
63 articleSubmission = (await articleSubmissionService.getAllSubmissions())
       [0];
64 t.is(articleSubmission.articleSubmissionState, ArticleSubmissionState.
       CLOSED);
65 });
```

Figure 5.3: Integration test 2/2: It tests the whole process from author creation up to an article submission

EKA Blockchain Solutions GmbH. They already have a strong background in the academic world and further they do understand the basics of Blockchain technology. During the development of the EUREKA platform, they have been strongly involved in discussions regarding the Editorial assessment and peer-review processes. They suggested potential solutions and continuously monitored the progress of the platform.

- *Second Group - People working for other Companies in the Blockchain Space*
  The second group consists of five people working for three different blockchain Start-Ups. All these people work for companies that are residents at Trust Square [73], the world's biggest blockchain hub in the heart of Zurich. Since the EKA Blockchain Solutions GmbH is a resident at Trust Square as well, it has been easier to connect with other members coming from the blockchain community. In this category, there are mainly people who have a very strong blockchain background but don't possess any know-how in the scientific field.

- *Third Group - People who are not part of the Blockchain community*
  The third category is composed by four external people and can be viewed as a small portion of the common user crowd. None of the members in this group possess prior knowledge in either blockchain technology or in the academic field.

We executed the SUS experiment after our first release in January 2019. The results are described in the section below and can be found in Figure 5.4.

### 5.2.1 Results

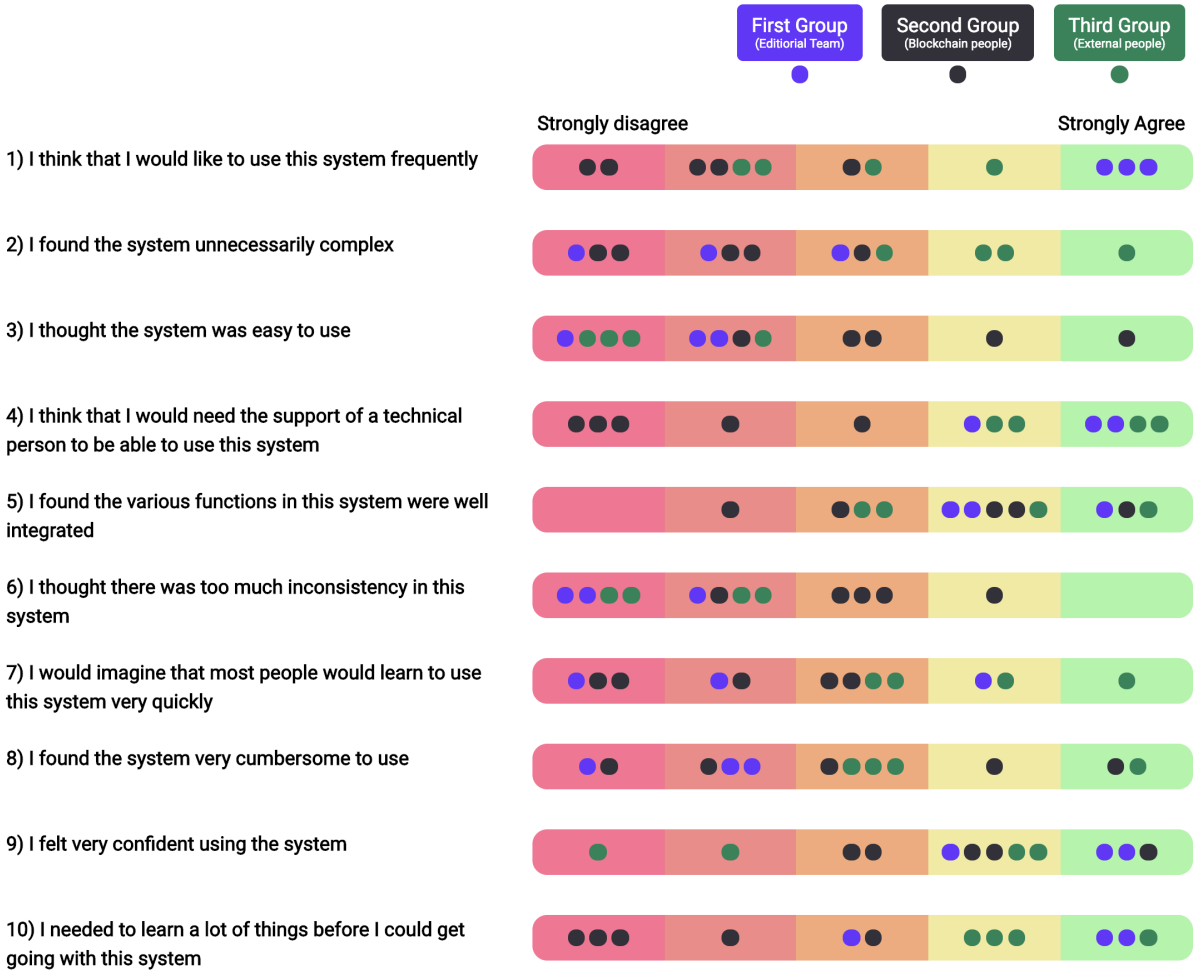



Figure 5.4: The System Usability Scale Template

In general, we can state that all groups considered the different functions in the EUREKA platform well integrated and free of inconsistency. Furthermore, almost all members felt very confident using the system, independently of the group to which they belong. On the one hand, people from the first and third group who do not possess any prior knowledge in the blockchain field, found the EUREKA platform difficult to use and stated that they needed to learn a lot of things before get going with the platform. In addition to that, based on the questionnaire we can state that they would need technical support to be able to interact with the system. On the other hand, members belonging to the blockchain community considered the platform uncomplicated and would not need any external help for using the system. None of these members needed to learn additional stuff before they were able to use the platform. It is worth to point out, that people from the third group without any know-how in either the scientific and blockchain field considered the system unnecessarily complex and strongly agreed to the fact that the platform could not be used without a technical person. Concerning the frequency usage of the system, only the Editorial team strongly agreed to the fact that they will use the

platform frequently. This is in accord with their field of interests. Based on the results, there exist substantial differences between people with and without blockchain expertise. On one side, our effort to build a user-friendly and easy-use blockchain-based platform has been highlighted by the answered given by the second group. In fact, most of them found the EUREKA platform intuitive and uncomplicated. On the other side, the current state of development and mass adoption of blockchain technology reflects the answers given by the first and third group. Indeed, most of them considered the required interactions with the blockchain complicated and still too technical.

# Chapter 6

# Summary and Conclusions

The academic industry is facing challenges and numerous scientists dispraise the current models for scholarly publishing. The limitations and high cost for scientist affecting constantly the scientific progress and thus the whole society.
As scientific achievements impact everybody's daily life, we consider it crucial to change the oligarchic process of publishing science.
In this sense, the blockchain technology is a possible new approach to change this. It has been taken into consideration as suitable options for addressing the documented problem of absence of fair credit contribution for the main stakeholders in the editorial and peer reviewing process in publishing science. We believe that this technology has the power to impact and change the daily life through new token-based incentive mechanism which can be self-regulated. With this new ecosystems scientific publishing could be improved and let the absence of rewards vanish by dentralizing the decision-making process.

The Eureka platform has been presented in this work as a blockchain-based MVP as alternative to the current, established models. With its goal to reward all involving parties fairly by distributing the EKA, open new perspectives for the faster and fairer scientific publish process. Eureka exploits the characteristics of a puplic blockchain for improving transparency and giving the possibility to everybody in the world to contribute and at the same time be fairly rewarded.
The amount paid in Eureka Token EKA for submitting a manuscript or a single observation is splitted and kept locked within a SC. Eventually it is proportionally and fairly distributed among those stakeholders who contributed in the process which lead an article to publication. Therefore Eureka avoids the well-known problem of reviewers not spending the amount needed for properly reviewing the quality of research. Similarly the editor, who invest time for the quality control of each research are beeing automatically paid when a article gets published. Further referenced authors, which indirectly contributed to a study are rewarded as well within the Eureka System.
Regarding the system designs, infrastructure and implementation related aspects, Eureka is build on the basis of three main components. A frond-ent component which acts as an interface for interacting with the Ethereum blockchain. A convenience-layer as back-end application which is constantly listening to the SC, monitoring it and providing an interface off-chain and lastly the SC itself. The entire architecture relies on state-of-arts web

technologies.

Eureka's main goal is to have a science publishing platform in a totally self-regulated ecosystem where the entire decision-making process is driven by the scientific community. This article presented our MVP as a first draft to prove the practicability of the idea. It lays a good starting point for developing this platform and bringing the product eventually to market.

# Chapter 7

# Installation Guide

The following instructions guide you to setup the platform on your local computer. All this installation steps can also be found in the README file of the official GitHub repository:

`https://github.com/eureka-blockchain-solutions/eureka-platform`

## 7.1 Prerequisites

- **Node.js [27]:** Install node.js on your local computer; version: ^10.0
  `https://nodejs.org/en/download/`
- **Yarn:** Install yarn on your local computer; version: ^1.13.0
  `https://yarnpkg.com/lang/en/docs/install/`
- **MongoDB [29]:** Install MongoDB on your local computer; version: ^4.0.3
  `https://www.mongodb.com/download-center/community`
- **Ganache [16]:** Install Ganache on your local computer; version: ^1.2.2
  `https://truffleframework.com/ganache`
- **Chrome Browser** Install Chrome Browser on your local computer; version: ^72.0
  `https://www.google.com/chrome/`
- **Meta Mask (Chrome extension)** Install Meta Mask as Chrome extension; version: ^5.3.1
  `https://metamask.io/`
- In addition we recommend a GUI for MongoDB to better work with the DB, e.g. Robo 3T: `https://robomongo.org/download`

## 7.2 Platform Setup

- Make sure all the prerequisites are prior installed

- Run the following:

```
$ git clone git@github.com:eureka-blockchain-solutions/eureka-platform.git
$ cd eureka-platform
$ yarn
```

- Create and start MongoDB locally:
  - Create directory at:

    ```
    > Windows:  'C:\data\db'

    > Unix:     'data/db'
    ```

  - Start mongoDB service:

    ```
    $ mongod
    ```

  - Connect to the service in another console:

    ```
    $ mongo
    ```

  - Use a new DB:

    ```
    $ use eurekaDB
    ```

  - Create a new DB admin:

    ```
    $ db.createUser({
    user: "{DB_USER}",
    pwd: "{DB_ADMIN}",
    roles: [{
    role: "dbAdmin",
    db:"{DB_NAME}"
    }]
    })
    ```

  - Insert something into the DB to save the DB:

    ```
    $ db.testData.insert({"test": "test"})
    ```

  - Check created DB:

    ```
    $ show dbs
    ```

- Copy '.env-example' and name it '.env'
- Provide the following mandatory environment variables in the .env file:

```
$ DB_HOST=  #host of DB (default: localhost)
$ DB_NAME= # {DB_NAME} set before in the DB
$ DB_USER= # {DB_USER} set before in the DB
$ DB_PASSWORD= # {DB_PASSWORD} set before in the DB
```

- The following environment variables are not mandatory, but are needed to be able to use all functionalities implemented:

```
  $ AMAZON_SECRET_KEY=  #Amazon S3 Database Secret Key
(Used for saving/providing dynamically resized Images)
  $ AMAZON_ACCESS_KEY= #Amazon S3 Database Access Key
    (Used for saving/providing dynamically resized Images)
  $ SENDGRID_API_KEY= #Sendgrid API access key
    (Used for sending Mails to Platform Users)
  $ ETHEREUM_FULL_NODE_KOVAN= #Address of SC in the Kovan Testnet
    (only needed when testing on Kovan Test network)
```

## 7.3 Start application locally with local Ganache Network

One way to start the whole platform is by let the app and also the blockchain network run locally. For that we start up the back-end application, use ganache to simulate a ethereum network and have a local database running.
There is no login process through Meta Mask, that is why it is necessary to call the front-end application by the incognito mode in the Chrome browser. This setup is great for testing purpose, as everything is running fast and local, which makes it easy to follow each step happening and discover bugs.

**How to start:**

- Start the database in a console

```
$ mongod
```

- Start Ganache application
- Assure that Ganache is running on port 7545:

```
Click on the gear wheel in the right corner of Ganache application.
Under Port number => type 7545.
```

- Deploy contracts

```
$ yarn run deploy-contracts
```

- Start the back-end application locally with a preloaded scenario to interplay with

```
$ yarn run scenario
```

- Open new console, start the front-end application

```
$ yarn run dev-client
```

- Open the chrome browser in the incognito mode and open the corresponding port (default port: `[http://localhost:3000/`)
- You should see the starting page of our platform.
- You can register yourself, by providing only your Email address (Authentication process was described in detail in Section 4.3.1)

## 7.4 Application interacting with kovan network

The SC was also published to the Kovan network by us. The platform is not totally tested in the environment of the Kovan network. Or ethereum node instance is originally hosted on digital ocean. Due to the restructuring of the company, the services of digital oceans are not used anymore, which means that the back-end application and with it all functionalities like monitoring, updating and time-based reverting are not executed on the Kovan network.
In order to be able to interact with the platform a user needs an active ethereum node on the Kovan network. He must provide the credentials to Meta Mask, so he can play with the Eureka platform. It is still possible to let the client application interact with the SC hosted on the Kovan Network, even if the functionalities are for now very limited. The following describe the procedure to start the local client, connecting to the Kovant Testnetwork.

**How to start:**

- Adapt the environment variables in the .env file

```
ETHEREUM_FULL_NODE_KOVAN=104.248.29.26:8546
BC_NETWORK=kovan
```

- Open new console, start the front-end application

```
$ yarn run dev-client
```

- Open the chrome browser on local port: `[http://localhost:3000/`)
- You should see the starting page of our platform.
- Open your Meta mask extension application and change to the Kovan Test network, as shown in Figure 7.1

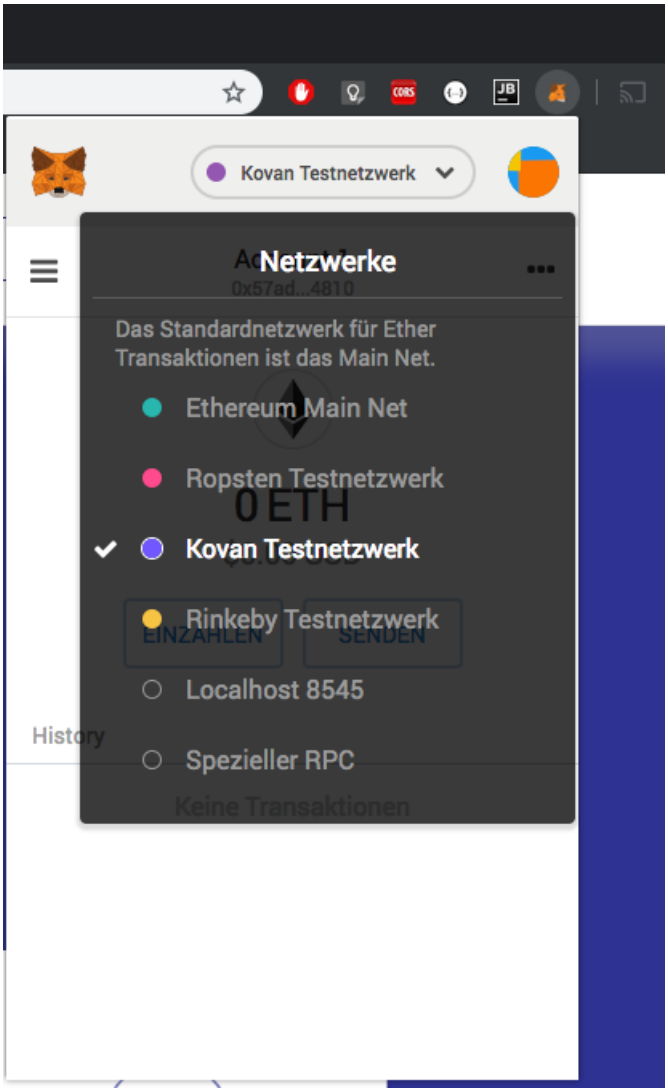



Figure 7.1: Changing to Kovan Testnetwork

- Make sure you are logged in in Meta Mask with your Kovan credentials
- You see the starting screen, but with a 'Kovan' Label as in Figure 7.1, informing the user, that he is using the Kovan Network

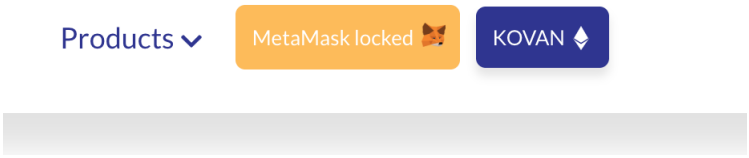



Figure 7.2: Kovan Label on the Starting page

- You can register yourself, by providing only your Email address (Authentication process was described in detail in Section 4.3.1)

# Chapter 8

# How to modify the platform

In this chapter we provide a manual how to make changes to the EUREKA platform. The change effects all the main parts of the architecture and is therefore a good modification sample.

**The modification scenario:**

We want to add a field to the document editor which the author of an article can use to specify a minimum score. This minimum score needs to be reached during the peer-review process, otherwise the article will not be published. An author may want to reach a minimum score for an article although the journal would publish it with a lower score.

**The modification scenario on GitHub:**

The code modifications were committed to a dedicated branch called *new-field-manual* which can be found in the GitHub repository:

`github.com/eureka-blockchain-solutions/eureka-platform/tree/new-field-manual`

All commit details can be seen by clicking on the commit SHA string at the begin of every section.

## 8.1 Initial situation

As already mentioned, we want to add a *minimum score* field to the document editor. Figure 8.1 shows the initial situation of the document editor. We want to add an input field after *Linked Articles* where we can specify our minimum score.

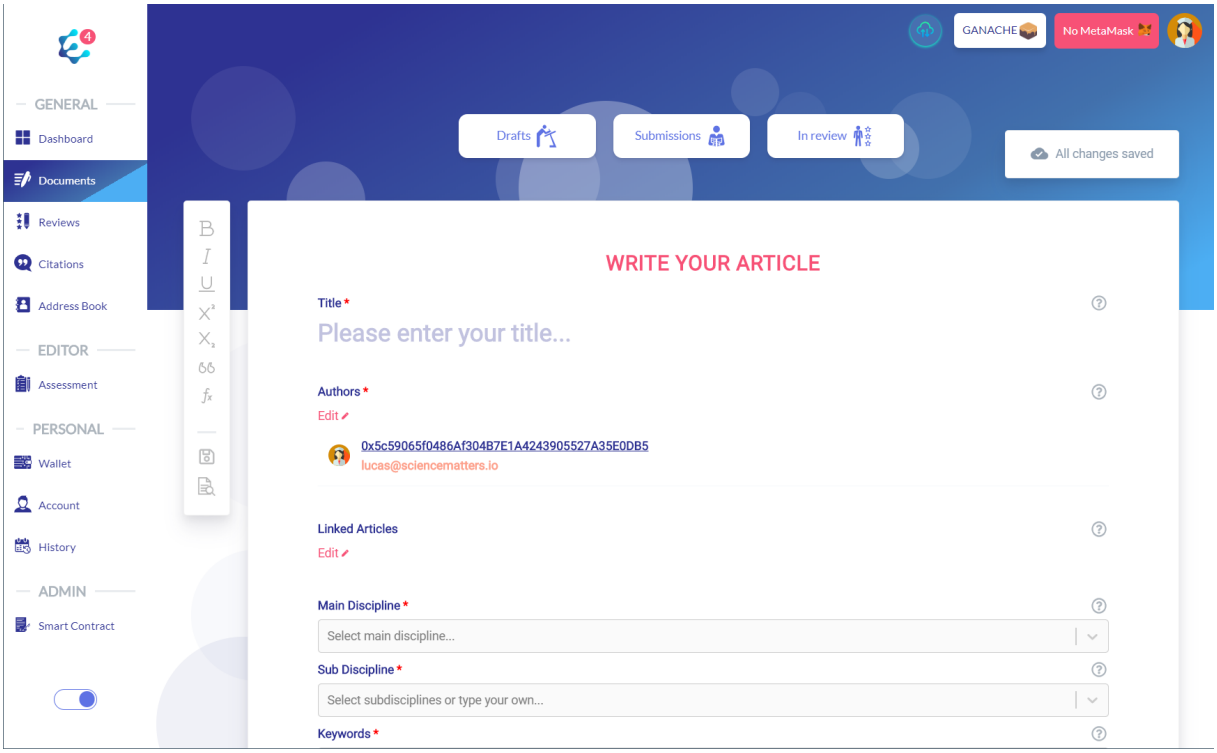



Figure 8.1: The document editor without the field minimum score to publish which we want to add after the line *Linked articles*

## 8.2 Step 1: Modify front-end by adding input field

This step can be seen in the commit *add minScore field in front-end* at `0af082ad`.

We create a new React component called `MinScoreField` and save it at *src/frontend/webpack/components/Articles/Online/TextEditor/MinScoreField.js*. It contains basically an input field with placeholder and a title above it.

In addition to that we open *src/frontend/webpack/components/Articles/Online/TextEditor/DocumentEditor.js* and add a `MinScoreField` component within a new `<Line>`. We add a minScore state variable and create a `onScoreChange` method which handles input changes.

```
<Line>
  <MinScoreField
    minScore={this.state.minScore}
    onScoreChange={score => {
        this.onScoreChange(score);
    }}
  />
</Line>
```

Listing 8.1: The new line added to DocumentEditor.js

The changes made can be seen in figure 8.2.

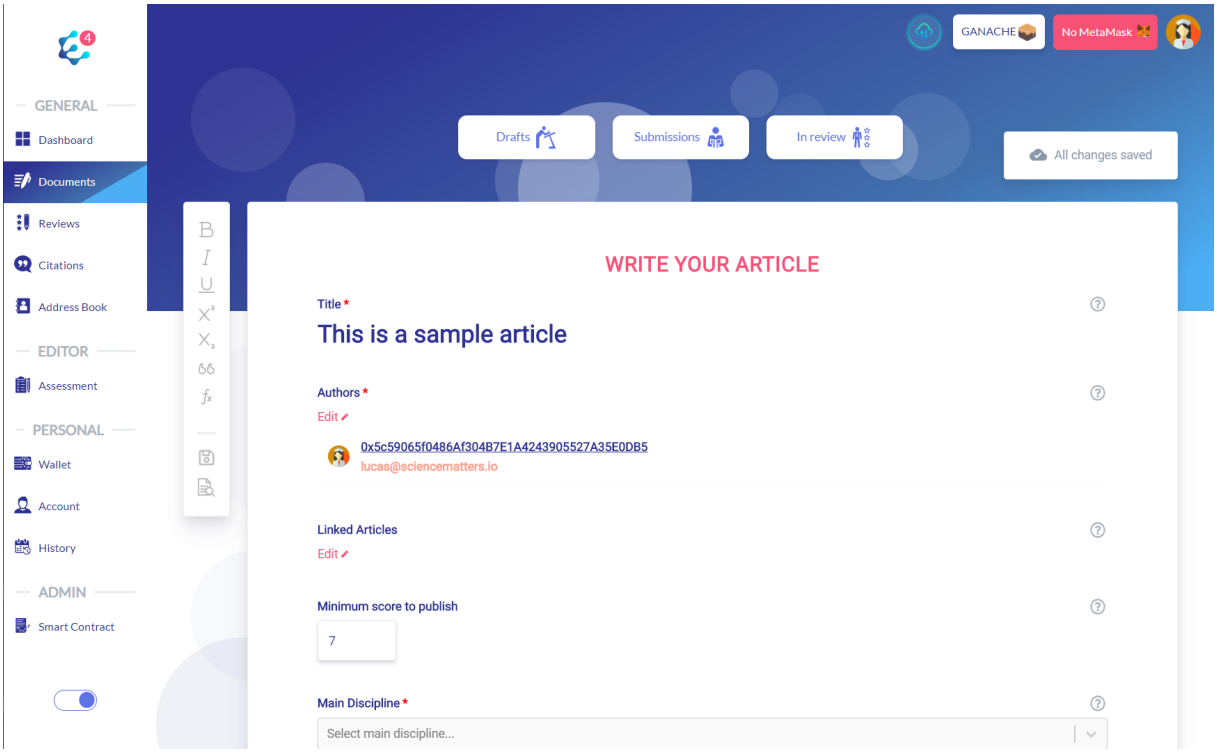



Figure 8.2: The document editor with the field minimum score to publish which we added to the front-end.

## 8.3 Step 2: Add the variable to the database schema

This step can be seen in the commit *add minScore variable to database schema* at `7cfeb357`.

Every database collection has a schema. We want to save an additional variable to the article versions collection, therefore we need to change the `articleVersionSchema` in *src/backend/schema/article-version.mjs*. We just add the following three lines:

```
1 minScore: {
2   type: Number
3 }
```

Listing 8.2: We add three lines to the article version schema

## 8.4 Step 3: Adjust back-end REST API and corresponding calls from front-end

This step can be seen in the commit *send minScore value to saveArticle methods -> adjust route and db service method* at `db16fbc2`.

Since we want to save the minimum score field to the back-end when writing a draft, we

have to pass `this.state.minScore` as parameter to the `saveArticle` method in *src/frontend/webpack/components/Articles/Online/TextEditor/DocumentMainMethods.js*. There, we have to pass the parameter to the body of the fetch-call.

In addition to the call adjustments, we also need to change the endpoint: We change the `PUT api/articles/:draftId` endpoint in *src/backend/routes/article-drafts-routes.mjs* such that `req.body.minScore` is passed to `articleVersionService.updateDraftById(...)`.

We then obviously have to align the changes in the *updateDraftById* method in *src/backend/db/article-version-service.mjs* and save minScore to the articleVersion object:

```
articleVersion.minScore = minScore;
```

## 8.5 Step 4: Adjust article data fetching in front-end

This step can be seen in the commit *adjust fetchArticle method: save minScore value from db to state and fill field value* at `e2e59019`.

Since we changed the article version schema already, the front-end gets the additional minScore variable automatically, without changing anything at the fetchArticle method (GET). We just have to read `response.data.minScore` and write it to the minScore-state (listing 8.3, line 11).

```
1  fetchArticle(draftId)
2    .then(response => response.json())
3    .then(response => {
4      if (response.success) {
5        (...)
6        this.setState({
7          _id: response.data._id,
8          document: deserialized,
9          lastSavedVersion: deserialized,
10         linkedArticles,
11         minScore: response.data.minScore
12       });
```

Listing 8.3: We write response.data.minScore to the DocumentEditor state

We also need to adjust the MinScoreField component a little bit: We have to pass minScore to the inputField value-props: we add `value=this.props.minScore` to the InputField-props.

## 8.6 Step 5: Adjust the affected structs in the EUREKA platform contract

This step can be seen in the commit *adjust EurekaPlatform contract: add minScore to ArticleVersion struct* at `d008b512`.

We want to save the minScore value to the platform contract's article version struct. Therefore we add a variable to the ArticleVersion struct. For a score value between 1 and 10 the uint8 data type is enough.

## 8.7 Step 6: Adjust the affected smart contract methods

This step can be seen in the commit *adjust EurekaPlatform contract: add minScore to ArticleVersion struct* at `d008b512`.

We also have to adjust all the smart contract methods that we can save the minScore value to the article version struct. For the moment, we start with the *startSubmissionProcess* method which needs to have an additional parameter `uint8 _minScore`.

## 8.8 Step 7: Adjust the smart contract call

The smart contract call to submit an article is a special and the most complicated call. The reason for it is that it is connected with a token transfer (submission fee transfer) and therefore needs to call the token contract first. Every other platform contract interaction is more straight forward and quite simple.

For simpler smart contract calls where the method is called directly we would only need to make sure that the correct value is passed to the new method parameter.

### 8.8.1 Adjust method hash representation

This step can be seen in the commit *startSubmissionProcess method has new hash representation -> update in transferAndCall method* at `94708313`.

Since the submit article method is connected with the transfer respectively the transferAndCall method of the token contract, we have to maintain this connection:

We adusted the parameters of the startSubmissionProcess method in the platform contract, therefore the hash representation of the method has changed too. The new hash representation is `0x90b0397c`. We change it in the submitArticle method in *src/smartcontracts/methods/token-contract-methods.mjs* (line 5). The parameters of the transferAndCall method need to be adjusted.

```
1 export const submitArticle = (_contract, _to, _amount, _data) => {
2   return _contract.methods.transferAndCall(
3     _to,
4     _amount,
5     '0x90b0397c',
6     _data
```

```
7   );
8 };
```

Listing 8.4: The new submitArticle method which calls the transferAndCall method

### 8.8.2 Adjust the input data state and the parameters of the called method

This step can be seen in the commit *add minScore parameter to submitArticle method* at `b1208a17`.

We change the computeInputData method, such that `this.state.minScore` is filled into `this.state.inputData.minScore`.

### 8.8.3 Change byte code representation of method parameters

This step can be seen in the commit *add minScore in articleHex calculation to path to transferAndCall method* at `c5ec493a`. A little correction which is explained below is made at `46ab2b01`.

The transferAndCall method of the EUREKA token contract can call any method on any smart contract. One just has to specify the method hash representation and pass the parameters encoded in one bytes object. Because we changed the startSubmissionProcess method's parameters (the method which is called from the token contract), we have to change this byte code calculation of the parameters.

The byte code calculation of the parameters can be found and changed in *src/smart-contracts/methods/get-articleHex.mjs*. We add a 32 byte hex representation of the minScore value to the bytecode at the slot after the article url (line 34) because we added the parameter in startSubmissionProcess after the article url too (see `46ab2b01`):

```
+ web3.utils.padLeft(web3.utils.toHex(minScore), 64).substring(2);
```

We also have to add 32 bytes to the arrayDataStartByte variable because the arrays are referenced first and then attached at the end of the bytecode.

Important to notice is that although we state in the smart contract that the minScore value is data type uint8 (one byte) only, the input data byte code always allocates 32 bytes for every parameter. This is why the changes in commit `c5ec493a` did not work.

Listing 8.5 shows the getArticleHex method after the changes:

```
1 const getArticleHex = (web3, article) => {
2   let amountOfArrays = 4;
3   let arrayDataStartByte =
4   0
5   // address msg.sender
6   + 32
```

```
7    // uint256 value
8    + 32
9    // bytes32 articleHash
10   + 32
11   // bytes32 url
12   + 32
13   // uint8 minScore variable (although it is only uint8 it is always
        passed with with a 64 digits hex string = 32 bytes)
14   + 32
15   //length of array references
16   + amountOfArrays * 32;
17
18   /*
19   connect the hex strings
20   */
21   let dataInHex =
22   '0x'
23   // articleHash
24   + article.articleHash
25   // URL
26   + web3.utils.padRight(web3.utils.toHex(article.url), 64).substring(2)
27   // minScore variable
28   + web3.utils.padLeft(web3.utils.toHex(minScore), 64).substring(2);
29
30   // reference author array
31   dataInHex += web3.utils
32   .padLeft(web3.utils.toHex(arrayDataStartByte), 64)
33   .substring(2);
34   arrayDataStartByte += 32 * (article.authors.length + 1);
35
36   (...)
37
38   // authors address array
39   dataInHex += web3.utils
40   .padLeft(web3.utils.toHex(article.authors.length), 64)
41   .substring(2);
42   article.authors.forEach(address => {
43     dataInHex += web3.utils.padLeft(address, 64).substring(2);
44   });
45
46   (...)
47
48   return dataInHex;
49 };
```

Listing 8.5: The getArticleHex method after the changes. This method encodes the startSubmissionProcess parameters into a byte code object.

## 8.9 Step 8: Additional smart contract adjustments and change event return values

This step can be seen in the commit *add minScore variable to submitArticleVersion method* at `4eb6cac4`.

To this point we just changed the smart contract call from the front-end. This interface is now set up again. We now have to make sure that the additional minScore parameter is used somehow and saved to the article version struct.

In this commit we pass the minScore value from the startSubmissionProcess method to the submitArticleVersion method because this is the method where the article version struct is created.

The next step can be seen in the commit *save minScore to ArticleVersion struct and emit it in the event* at `a6e640b1`.

Here we actually change the submitArticleVersion and save the minScore value to the article version: `article.minScore = _minScore;`. We also change the SubmissionProcessStart event. This event is emitted when the changes where successful. The emits includes an additional return value now: the minScore value.

## 8.10 Step 9: Adjust the platform contract monitor

This step can be seen in the commit *read minScore from emitted event and save to database object* at `01205b40`.

This is the last step of the modification. When we add an additional variable to the smart contract, we probably want to mirror that to the database. This is the reason why we adjusted the event return values in the previous step. Now we just have to read this additional value and write it to the database. You probably ask yourself why this step is needed because we already wrote this value to the database. Imagine if this is a value which can be changed from another user potentially not writing it to the database first: the smart contract would be changed but the database would not reflect the smart contract state anymore.

Therefore we read the event return values in the SubmissionProcessStart event listener in *src/backend/web3/contract-event-lister.mjs* (`event.returnValues.minScore`) and pass it to the updateSubmissionStartByArticleHash method where we actually update the minScore value from the database with the one from the smart contract.

# Bibliography


[1] EUREKA: The next-generation open-science platform powered by blockchain, https://eurekatoken.io/ (Last accessed January 2019).

[2] ScienceMatters: The next-generation publishing platform, https://sciencematters.io/ (Last accessed January 2019).

[3] ReactJS: A JavaScript library for building user interfaces, https://reactjs.org/ (Last accessed January 2019).

[4] Agiliq: Understanding State and Props in ReactJS, https://www.agiliq.com/blog/2019/05/understanding-react-state-and-props/ (Last accessed January 2019).

[5] W3Schools: HTML Styles - CSS, https://www.w3schools.com/html/html_css.asp (Last accessed January 2019).

[6] Web3.js: Ethereum JavaScript API, https://web3js.readthedocs.io/en/1.0/index.html (Last accessed January 2019).

[7] Web3.js Callbacks Promises Events, https://web3js.readthedocs.io/en/1.0/callbacks-promises-events.html (Last accessed February 2019).

[8] GitHub, Eureka-Blockchain-Solutions: Front-end, https://github.com/eureka-blockchain-solutions/eureka-platform/tree/master/src/frontend (Last accessed January 2019).

[9] W3Schools: The Window Object, https://www.w3schools.com/jsref/obj_window.asp (Last accessed January 2019).

[10] Ethereum: Blockchain App Platform, https://www.ethereum.org/ (Last accessed January 2019).

[11] Ethereum: Create your own crypto-currency, https://www.ethereum.org/token (Last accessed February 2019).

[12] Github Ethereum/EIPs - ERC: transferAndCall Token Standard, https://github.com/ethereum/EIPs/issues/677 (Last accessed February 2019).

[13] Github Ethereum/EIPs - ERC865: Pay transfers in tokens instead of gas, in one transaction, https://github.com/ethereum/EIPs/issues/865 (Last accessed February 2019).

[14] Github OpenZeppelin/openzeppelin-solidity - SafeMath.sol, https://github.com/OpenZeppelin/openzeppelin-solidity/blob/c63b203c1d1800c9a8b5f51f0b444187fdc6c185/contracts/math/SafeMath.sol (Last accessed February 2019).

[15] MetaMask: Brings Ethereum to your browser, https://metamask.io/ (Last accessed January 2019).

[16] Ganache: One Click Blockchain, https://truffleframework.com/ganache (Last accessed January 2019).

[17] INFURA: Your access to the Ethereum network, https://infura.io/ (Last accessed January 2019).

[18] Draft.js: Rich Text Editor Framework for React, https://draftjs.org/ (Last accessed January 2019).

[19] EditorState: Top-level state object for Draft.js , https://draftjs.org/docs/api-reference-editor-state (Last accessed January 2019).

[20] SHA-256 Cryptographic Hash Algorithm , https://www.movable-type.co.uk/scripts/sha256.html (Last accessed January 2019).

[21] Introducing JSON: a lightweight data-interchange format , https://www.json.org/ (Last accessed January 2019).

[22] Canonical JSON , http://wiki.laptop.org/go/Canonical_JSON (Last accessed January 2019).

[23] Wikipedia: Sanity Check, https://en.wikipedia.org/wiki/Sanity_check (Last accessed January 2019).

[24] Solidity Read the docs https://solidity.readthedocs.io/en/v0.4.25/ (Last accessed January 2019).

[25] Struct Types - Solidity Read the docs https://solidity.readthedocs.io/en/v0.4.25/types.html#structs (Last accessed January 2019).

[26] Contract ABI Specification - Solidity Read the docs https://solidity.readthedocs.io/en/develop/abi-spec.html#examples (Last accessed January 2019).

[27] Node.js: a script runtime built https://nodejs.org/en/ (Last accessed January 2019).

[28] Express: a web application framework for node.js https://expressjs.com/ (Last accessed January 2019).

[29] MongoDB: a document database with scalability and flexibility https://www.mongodb.com/ (Last accessed January 2019).

[30] MongooseJS: elegant mongodb object modeling for node.js https://mongoosejs.com/ (Last accessed January 2019).

[31] Sendgrid: Send Newsletter & Promotional Emails with confidence `https://sendgrid.com/` (Last accessed January 2019).

[32] PassportJs: Simple, unobtrusive authentication for Node.js `http://www.passportjs.org/` (Last accessed January 2019).

[33] Passport npm module: NPM package for passport.js, compatible with node.js `https://www.npmjs.com/package/passport` (Last accessed February 2019)

[34] Redux Framework: A JavaScript predictable state container `https://redux.js.org` (Last accessed January 2019).

[35] Elsvier: Empowering Knowledge.`https://www.elsevier.com/` (Last accessed January 2019).

[36] IEEE Explore: IEEE Explore Digital Library. `https://ieeexplore.ieee.org/` (Last accessed January 2019).

[37] Springer Link. Providing Researchers With Access to Millions of Scientific Documents from Journals, Books, Series, Protocols and Reference Works. `https://link.springer.com/` (Last accessed January 2019).

[38] International Association of Scholarly Publishing's market report from 2015. `https://www.stm-assoc.org/2015_02_20_STM_Report_2015.pdf` (Last accessed January 2019)

[39] EUREKA Whitepaper - A peer-to-peer scientific data publishing platform powered by Blockchain, to promote open science `https://eurekatoken.io/whitepaper_3_1.pdf` (Last accessed January 2019)

[40] Monya Baker - 1500 scientists Lift The Lid on Reproducibility `https://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970` (Last accessed January 2019).

[41] React Router: Basic Routing `https://reacttraining.com/react-router/web/guides/quick-start` (Last accessed January 2019).

[42] Stateful vs. Stateless Functional Components in React `https://code.tutsplus.com/tutorials/stateful-vs-stateless-functional-components-in-react--cms-29541` (Last accessed January 2019).

[43] Documentation for Styled Components `https://www.styled-components.com/docs/basics` (Last accessed January 2019).

[44] Redux Actions `https://redux.js.org/basics/actions` (Last accessed January 2019).

[45] Redux Reducers `https://redux.js.org/basics/reducers` (Last accessed January 2019).

[46] Redux Store `https://redux.js.org/basics/store` (Last accessed January 2019).

[47] Why use Redux? Reasons with clear examples https://blog.logrocket.com/why-use-redux-reasons-with-clear-examples-d21bffd5835 (Last accessed January 2019).

[48] Redux DevTools https://github.com/reduxjs/redux-devtools (Last accessed January 2019).

[49] Orvium: The Open Source and Decentralized Framework for Managing Scholarly Publications Life Cycles and The Associated Data. https://orvium.io (Last accessed January 2019)

[50] Responsive Web Design: Media Queries https://www.w3schools.com/css/css_rwd_mediaqueries.asp (Last accessed January 2018).

[51] ScienceRoot https://www.scienceroot.com (Last accessed January 2019).

[52] Blockchain for Open Science https://www.blockchainforscience.com (Last accessed January 2019).

[53] What is DAO? https://blockchainhub.net/dao-decentralized\-autonomous-organization (Last accessed January 2019).

[54] Blockchain for Open Science - The living document. https://www.blockchainforscience.com/2017/02/23/blockchain-for-open-science-the-living-document/ (Last accessed January 2019).

[55] IPFS is The Distributed Web. https://ipfs.io/ (Last accessed January 2019).

[56] Building a New Home for Science https://deip.world/ (Last accessed January 2019).

[57] Sendgrid/mail: Mail Service for the Sendgrid v3 Web API https://www.npmjs.com/package/@sendgrid/mail (Last accessed January 2019).

[58] Cron: Allows you to execute something on a schedule https://www.npmjs.com/package/cron (Last accessed January 2019).

[59] Introducing JSX https://reactjs.org/docs/introducing-jsx.html (Last accessed January 2018).

[60] Argon Design System https://demos.creative-tim.com/argon-design-system/index.html?_ga=2.268656502.625706937.1549100533-387788188.1536925437 (Last accessed February 2018).

[61] NPM: The package manager for JavaScript and the world's largest software registry https://www.npmjs.com/ (Last accessed February 2019).

[62] Template literals (Template strings) https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals (Last accessed February 2019).

[63] Binary Operator https://www.techopedia.com/definition/23953/binary-operator (Last accessed February 2019).

[64] Destructuring assignment https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment (Last accessed February 2019).

[65] F1000research: Publish all your findings including null results, data notes and more https://f1000research.com/ (Last accessed February 2019).

[66] Digital Ocean Droplets: Scalable compute services https://www.digitalocean.com/products/droplets/ (Last accessed February 2019).

[67] Kovan Testnet: The Fast and Reliable Ethereum Test Chain https://kovan-testnet.github.io/website/ (Lasr accessed February 2019).

[68] Remix IDE https://remix.readthedocs.io/en/latest/ and https://remix.ethereum.org/ (Last accessed February 2019).

[69] System Usability Scale (SUS) https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html (Last accessed February 2019).

[70] System Usability Scale Template http://www.uxforthemasses.com/wp-content/uploads/2011/04/SUS-System-Usability-Scale.doc (Last accessed February 2019).

[71] Aziz. Coins, Tokens, and Altcoins, What is The Difference? https://masterthecrypto.com/differences-between-cryptocurrency-coins-and-tokens/ (Last accessed December 2018).

[72] StrategicCoins. ICO 101: Utility Tokens vs. Security Tokens https://strategiccoin.com/ico-101-utility-tokens-vs-security-tokens/ (Last accessed December 2018).

[73] Trust Square, the world's biggest blockchain hub Theworld'sbiggestblockchainhub (Last accessed February 2019).

[74] Model-view-Controller, Architectural pattern https://en.wikipedia.org/wiki/Model-view-controller (Last accessed March 2019)

[75] Bcrypt package, NPM Package using the bcrypt algorithm https://www.npmjs.com/package/bcrypt (Last accessed March 2019)

[76] Bcrypt, a password hashing function https://en.wikipedia.org/wiki/Bcrypt (Last accessed March 2019)

[77] REST - Representational State Transfer: A software architectural style that defines a set of constraints to be used for creating Web services. https://en.wikipedia.org/wiki/Representational_state_transfer (Last accessed February 2019).

[78] Express-session package, NPM package used for creating sessions on a server https://www.npmjs.com/package/express-session (Last accessed March 2019)

# List of Figures


3.1 The general EUREKA Transaction Flow . . . . . 8

4.1 EUREKA Platform Process Model, platform-stakeholder interaction, part 1 13

4.2 EUREKA Platform Process Model, platform-stakeholder interaction, part 2 14

4.3 EUREKA's Architecture . . . . . 15

4.4 EUREKA's Colors.js vs. Index.css . . . . . 17

4.5 EUREKA's Front-End Architecture . . . . . 26

4.6 EUREKA's color schema . . . . . 28

4.7 Welcome Screen . . . . . 28

4.8 Login Screen . . . . . 29

4.9 Register Screen . . . . . 29

4.10 EUREKA's Dashboard Screen . . . . . 30

4.11 Author Drafts . . . . . 30

4.12 Author Submissions . . . . . 31

4.13 Online Editor - Upper Part . . . . . 31

4.14 Online Editor - Lower Part . . . . . 32

4.15 Online Editor - Smart Contract Input Data . . . . . 32

4.16 EUREKA's Address Book . . . . . 33

4.17 Editorial Assessment - Article Assignment . . . . . 34

4.18 Editorial Assessment - Article Assignment - Article Preview . . . . . 34

4.19 Editorial Assessment - Article Assignment - Blockchain Synchronization . . 35

4.20 Editorial Assessment - Article Assignment - Transaction Pool . . . . . 35
4.21 Editorial Assessment - Sanity Check . . . . . 36
4.22 Editorial Assessment - Sanity Check - Transaction Pool . . . . . 36
4.23 Editorial Assessment - Pool of Reviewers . . . . . 37
4.24 Editorial Assessment - Reviewers Invitation . . . . . 37
4.25 Editorial Assessment - Reviews Check . . . . . 38
4.26 Editorial Assessment - Article Finalization . . . . . 38
4.27 Peer Review - Invitation . . . . . 39
4.28 Peer Review - Annotation Tool . . . . . 39
4.29 User Account . . . . . 40
4.30 Blockchain Transaction History . . . . . 41
4.31 Smart Contract Owner Dashboard . . . . . 41
4.32 Mobile Version - Dashboard . . . . . 42
4.33 Mobile Version - Article Preview . . . . . 42
4.34 Mobile Version - Editorial Assessment - Article Assignment . . . . . 43
4.35 Object model of the current version of the platform contract . . . . . 45
4.36 Object model of the current version in the data-base . . . . . 61
4.37 The event-listener for the articleSubmission on the SC back-end side . . . . 62
4.38 The directory tree of the backend file structure . . . . . 63
4.39 user-routes.mjs: Routing example handling request on '/users/roles' . . . . 64
4.40 user-service.mjs: Service example managing interaction with the database . 64
4.41 user-mjs: Schema example for User collection in the database . . . . . 65
4.42 roles-enum.mjs: Enumeration example used for the User . . . . . 66
4.43 access-controller.mjs: Middleware for handling access control . . . . . 67
4.44 article-version-routes.mjs: Example Code snippet showing the use of rolesOnly 68
4.45 error-thrower.mjs: Example error messages as example code snipped . . . . 69
4.46 local-passport.mjs: Own defined Passport Strategy . . . . . 70
4.47 The local Ethereum blockchain provider Ganache . . . . . 77

4.48 The Ethereum Solidity IDE Remix after clicking ”Deploy” and before confirming the transaction with MetaMask . . . . . 79

4.49 contract-event-listener.mjs: Handling all events from SC in the backend . . 83

4.50 timebased-contract-service.mjs: Handling time-based triggers of SC-functions 84

4.51 timebased-contract-service.mjs: Get IDs of Editor with a time-out . . . . . 85

4.52 timebased-contract-service.mjs: Remove Editors from Submission on the SC 86

4.53 The sign-up-for-reviewing transaction in detail . . . . . 88

5.1 One of many integration tests, testing the proper interaction between smart-contract and back-end when an user gets editor-assigned . . . . . 91

5.2 Integration test 1/2: It tests the whole process from author creation up to an article submission . . . . . 92

5.3 Integration test 2/2: It tests the whole process from author creation up to an article submission . . . . . 93

5.4 The System Usability Scale Template . . . . . 95

7.1 Changing to Kovan Testnetwork . . . . . 103

7.2 Kovan Label on the Starting page . . . . . 103

8.1 The document editor without the field minimum score to publish which we want to add after the line *Linked articles* . . . . . 106

8.2 The document editor with the field minimum score to publish which we added to the front-end. . . . . 107