Lehrstuhl Angewandte Informatik IV
Datenbanken und Informationssysteme
Prof. Dr.-Ing. Stefan Jablonski

Institut für Angewandte Informatik
Fakultät für Mathematik, Physik und Informatik
Universität Bayreuth

Project Report

Philipp Scholz, Anatoly Obukhov

August 6, 2021
Version: Final

Universität Bayreuth

Fakultät Mathematik, Physik, Informatik
Institut für Informatik
Lehrstuhl für Angewandte Informatik IV

Blockchain-based Process Execution with Chrysalis

Project Report

Philipp Scholz, Anatoly Obukhov

1. Reviewer Prof. Dr.-Ing. Stefan Jablonski
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth

2. Reviewer Dr. Lars Ackermann
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth

Supervisors Christian Sturm and Lars Ackermann

August 6, 2021

Philipp Scholz, Anatoly Obukhov
Project Report
Blockchain-based Process Execution with Chrysalis, August 6, 2021
Reviewers: Prof. Dr.-Ing. Stefan Jablonski and Dr. Lars Ackermann
Supervisors: Christian Sturm and Lars Ackermann

Universität Bayreuth
Lehrstuhl für Angewandte Informatik IV
Institut für Informatik
Fakultät Mathematik, Physik, Informatik
Universitätsstrasse 30
95447 Bayreuth
Germany

Abstract

TODO Abstract

Contents

1 Introduction

> *You can't do better design with a computer, but you can speed up your work enormously.*
>
> — **Wim Crouwel**
> (Graphic designer and typographer)

1.1 Motivation and Problem Statement

1.2 Results

1.3 Thesis Structure

Chapter 2

In the first content chapter, we will explain the way Chrysalis generally functions. Starting from an abstract and high-level view where the intended uses of the application are explained, we will the continue to detail the components that make up Chrysalis - what their role in the system is, how they interact with each other and with what external components they interface. At last, we will describe how business processes are modelled inside a blockchain node, so that our application may interact with them in a defined way.

Chapter ??

This chapter being the biggest of all, we intend to describe our programming work done here. This includes all improvements, additions and removals in the code. To do this, for every major task we will first describe its meaning and implications, then give a modeler's overview of the changes done. Where needed, we will give some technical insights to our work. Concluding every task as well as the entire chapter, we will summarize our results.

Chapter 4

Given that this project wasn't intended to be perfected once our work was done, and

also given that some problems arose that hampered the quality of our results, this chapter will describe said issues. We will both clarify where they stem from and propose some ways of solving them, so those who will be handed this project may fix them with relative ease.

Chapter 5

TODO

Chapter 6

Due to the project having grown in complexity during our work on it, we decided to build a repository of design documents and other helpful files. In this short section, we intend to present these resources.

Chapter 7

Finally, we will evaluate the success of our project and speak about the opportunities and limits it offers to those interested in deploying a decentralized process management engine.

2 Architecture of Chrysalis

TODO Chrysalis is Javascript + NodeJS + React

2.1 Intended Usage

TODO Configuration, Deployment, Execution

2.2 Components and their Interactions

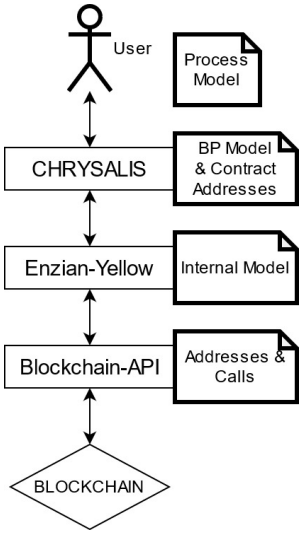

Fig. 2.1: Layer structure of the original *Chrysalis* System.

2.3 Modeling of Processes

TODO Introduce Smart Contracts and Data as primary Blockchain Components
TODO explain how Processes are represented with those.

3 Improvements

3.1 Restructuring the Code

TODO We'll focus on Enzian-Yellow here

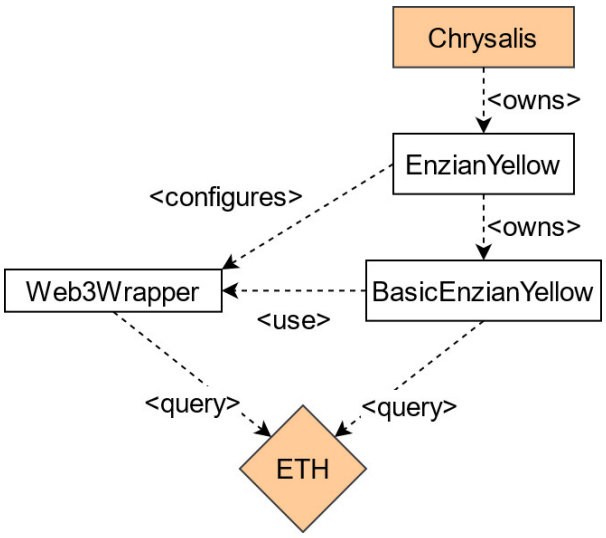

Fig. 3.1: Component structure of the original *Enzian-Yellow* Repository. The components marked in orange are not part of Enzian-Yellow, but interact with it.

TODO describe the general structure of the original Enzian-Yellow

3.1.1 Task Breakdown

3.1.2 Improving the Dependency Hierarchy

3.1.3 Interface for Expansions

3.1.4 Transparency

3.1.5 Minor Improvements

3.2 Persistence layer

In this section we will discuss implementation of the Persistence layer of CHRYSALIS. We will list the problems solved by this tasks, details of the implementation both on the front- and backend side and results achieved by it.

3.2.1 Problem statement

Initially, when we got our hands on the project, CHRYSALIS stored all of the off-chain configuration data in the browser's local storage. Not only is it a safety concern (since the frontend user can easily manipulate data however they want), but also it is not reliable, since the browser's local storage could be cleaned on the user side and all of the data would be lost.

The other point of concern was that the only way to access deployed process model was by explicitly typing in its contract address, which renders the user interface completely useless and ruins user experience. Furthermore, to pick a task to execute, one had to explicitly specify the id assigned to it after the parsing into enzian model, which the user might not have even noticed. Lastly, there was no constraints on the task identifiers that could be chosen, so the user was perfectly capable of choosing an incorrect one and getting an error. To combat that, it was decided to implement a persistence layer, which would store all the off-chain information in a database, including information on associations between tasks and deployed processes.

Fig. 3.2: Process execution page before implementation of the persistence layer

3.2.2 Software stack

To implement this functionality it was decided to build a separate REST-server. For the server's implementation express.js framework was chosen, since the entire project is in javascript and express.js is meant for RESTful API implementation. PostgreSQL was chosen as a database management system, mainly because it is widely used and optimized for production, but free at the same time (unlike, for example, Oracle). It also provides a wide array of object-relational functionality, which could be useful further down the line in CHRYSALIS development. For exchange between the server and the database Sequelize ORM is used.

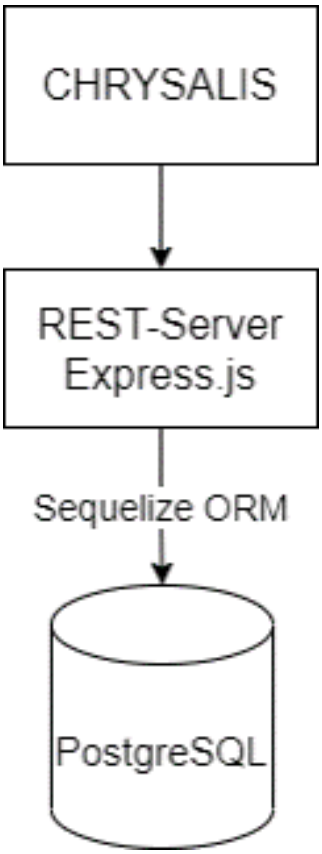

Fig. 3.3: Persistence layer architecture

3.2.3 Backend

As is required by Sequelize ORM and express.js framework, the server code is divided into four packages: models, migrations, controllers and routes. Models contain a representation of database entities, including column datatypes, constraints, associations and cardinalities. Migrations contain scripts used for propagating the database schema created in the model package and all the changes made in that schema to the database. Every migration contains a function for propagating the changes and a function for undoing them. The controller package contains the

server's business-logic, the database interactions in particular. The routes package provides REST-API endpoints for communication with the server.

The database schema is rather simple and consists of six entities (apart from the system ones, needed for the Sequelize ORM to work). Those entities are: Process, task, connection, abi, setting and account. The entities "Process" and "Task" are self-explanatory. Connections contain information about blockchain networks that the user could connect to. Account contains information regarding user's account on the blockchain network, such as their private key for signing transactions. Abi is an entity containing compiled smart contract code for executing a process model and is deployed every time a new process is uploaded to the system. The "Settings" entity contains current set of connection configurations, chosen by the user. Processes and Tasks are connected by a one-to-many association.

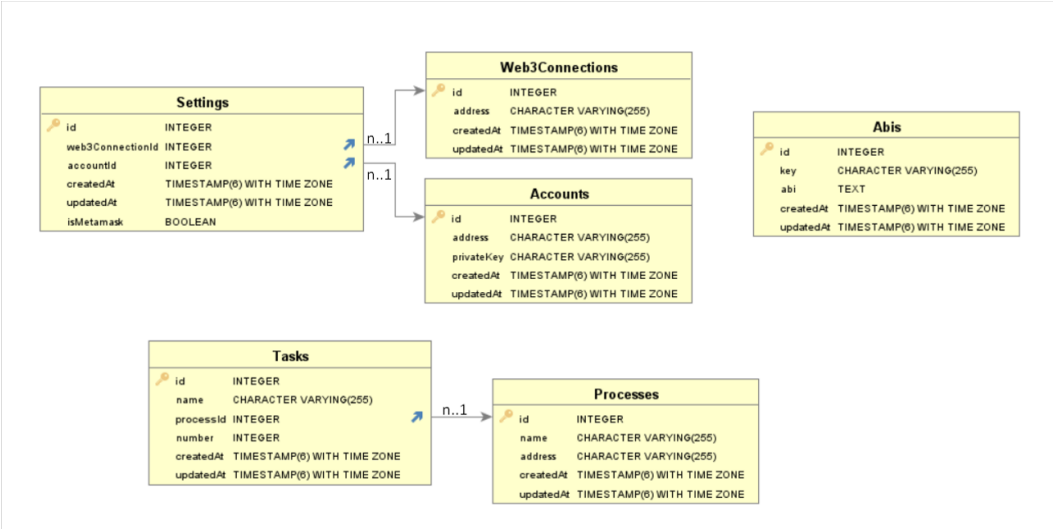

Fig. 3.4: Database schema

3.2.4 Frontend

On the frontend side we had to work within the confines of the existing application. The main means of RESTful exchange in react.js is Fetch-API. But the problem is that Fetch-API is very wordy and we would have to reuse large blocks lots of times, in every component of the application. Not only that, nut one would have to call this API with a wide array of different parameters, depending on the caller's intention. So it was decided to implement some universal exchange handling functionality within the frontend application.

To implement this it was decided to create an ExchangeHandler class which would be called by the application components to handle their requests. The components would pass to it the request method, URI and optionally data that they have to send, and then based on the method the exchange handler would find the appropriate instance of one of the sender classes and call them to execute the request. These sender classes are GetRequestSender, PostRequestSender, PatchRequestSender and DeleteRequestSender. They are instantiated and kept in the SenderRepository class. It contains a map with request methods as keys and sender instances as values. The exchange handler gets an appropriate sender from this map and calls its send() method to make a request to the server, returning a special promise objects, which provides methods to specify a logic, that should be executed once the response is received.

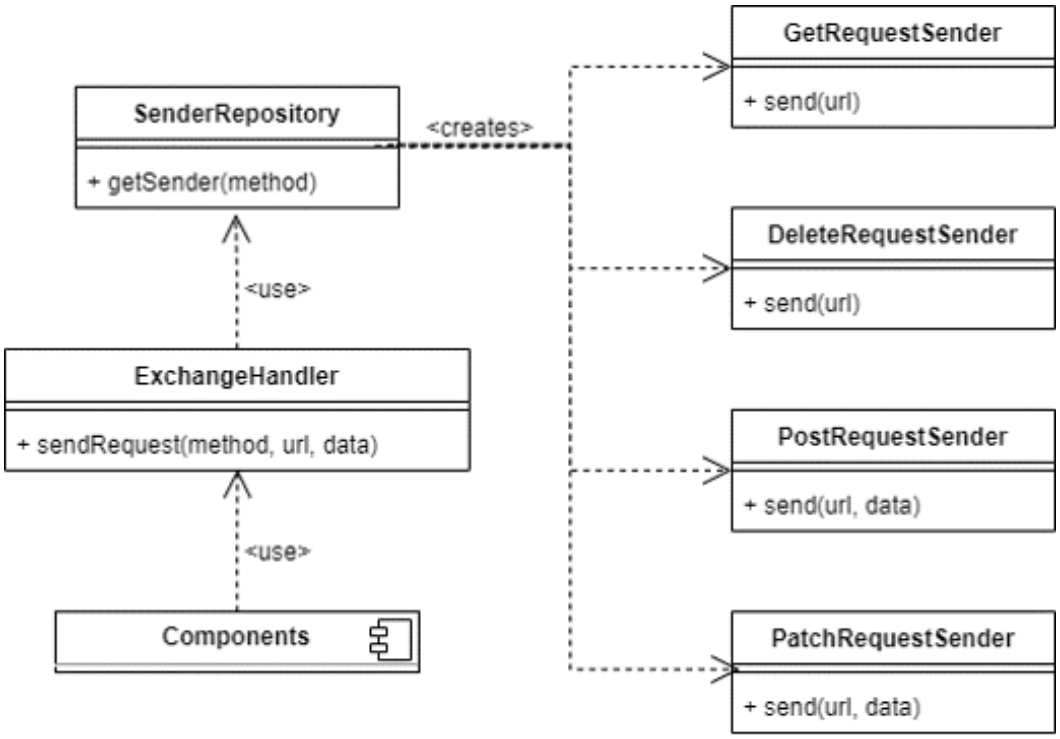

Fig. 3.5: Database exhange module

3.2.5 Result

After the changes made, all of the off-chain data was moved from the local storage to a separate database, improving safety and reliability. User experience was also significantly improved, since it became possible to choose deployed processes by their name from a list provided by the server, as well as choose from tasks associated with the process, by their name as well. The functionality of retrieving deployed processes by their contract addresses (if they are not present in the database) was preserved as well, but it underwent slight changes to make it compatible with

the new architecture, and these changes will be discussed in the smart contract optimization section.

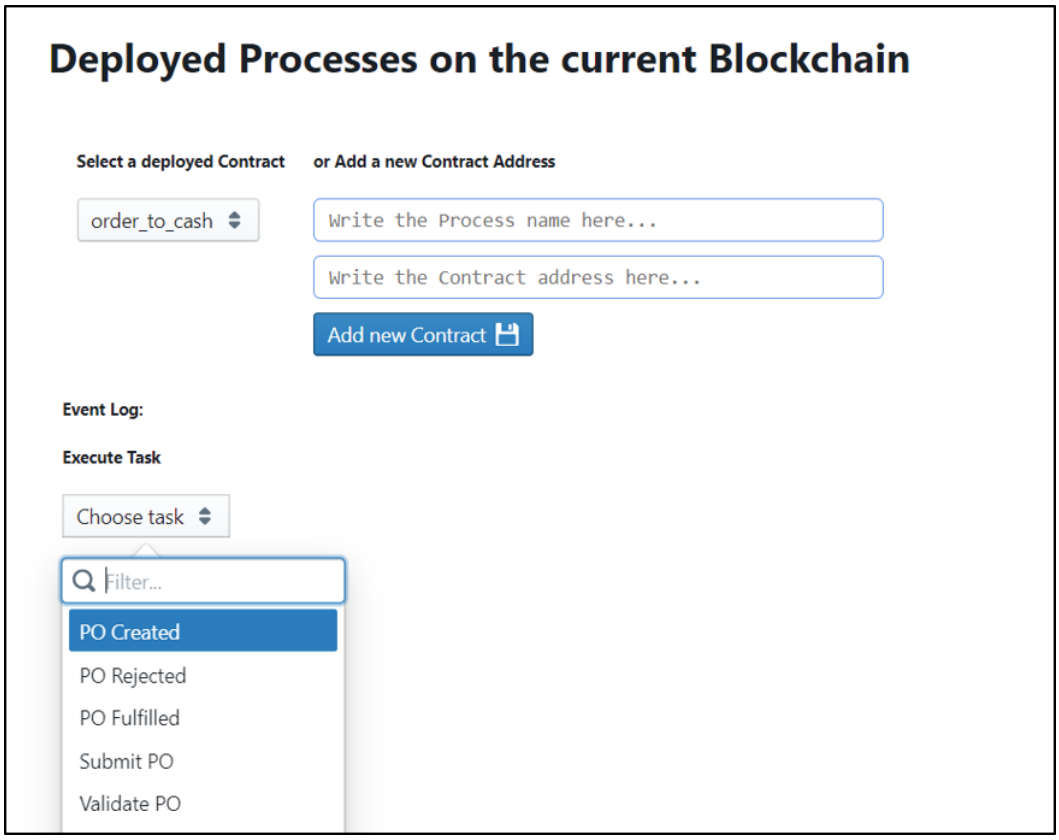

Fig. 3.6: Process execution page after the improvements

3.3 Hyperledger-based Application

TODO Task: Build HL application for Chrysalis

3.3.1 Hyperledger as a Ledger Protocol

3.3.2 Required Components

3.3.3 Test-Network

3.3.4 Representation of Processes

3.3.5 Process Deployment and Execution

3.3.6 Integration into Chrysalis

3.3.7 Result

3.4 ethereum overhaul

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Fig. 3.7: Figure example: *(a)* example part one, *(c)* example part two; *(c)* example part three

This is the second paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like

“Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

3.5 Summary of Improvements

4 Open Issues

Caterpillar

5

5.1 Caterpillar Section 1

Resources

6

> *Users do not care about what is inside the box, as long as the box does what they need done.*
>
> — **Jef Raskin**
> about Human Computer Interfaces

6.1 Resources Section 1

7 Conclusion

7.1 Conclusion Section 1

List of Figures

List of Tables

Declaration

You can put your declaration here, to declare that you have completed your work solely and only with the help of the references you mentioned.

Bayreuth, August 6, 2021

Philipp Scholz

You can put your declaration here, to declare that you have completed your work solely and only with the help of the references you mentioned.

Bayreuth, August 6, 2021

Anatoly Obukhov