



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

- |                    |   |
|--------------------|---|
| <i>1. Reviewer</i> | <b>Prof. Dr.-Ing. Stefan Jablonski</b><br>Fakultät Mathematik, Physik, Informatik<br>Universität Bayreuth |
| <i>2. Reviewer</i> | <b>Dr. Lars Ackermann</b><br>Fakultät Mathematik, Physik, Informatik<br>Universität Bayreuth              |
| <i>Supervisors</i> | 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

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Business Processes on Blockchain . . . . .	1
1.2	Problem Statement . . . . .	1
1.3	Results . . . . .	1
1.4	Thesis Structure . . . . .	1
<b>2</b>	<b>Architecture of Chrysalis</b>	<b>3</b>
2.1	Intended Usage . . . . .	3
2.2	Components and their Interactions . . . . .	4
2.3	Modeling of Processes . . . . .	5
<b>3</b>	<b>Improvements</b>	<b>7</b>
3.1	Restructuring the Code . . . . .	7
3.1.1	Task Breakdown . . . . .	7
3.1.2	Improving the Dependency Hierarchy . . . . .	7
3.1.3	Interface for Expansions . . . . .	7
3.1.4	Transparency . . . . .	7
3.1.5	Minor Improvements . . . . .	8
3.2	Persistence layer . . . . .	8
3.2.1	Problem statement . . . . .	8
3.2.2	Software stack . . . . .	9
3.2.3	Backend . . . . .	9
3.2.4	Frontend . . . . .	10
3.2.5	Result . . . . .	11
3.3	Hyperledger-based Application . . . . .	12
3.3.1	Hyperledger as a Ledger Protocol . . . . .	12
3.3.2	Required Components . . . . .	14
3.3.3	Test-Network . . . . .	14
3.3.4	Representation of Processes . . . . .	14
3.3.5	Process Deployment and Execution . . . . .	15
3.3.6	Integration into Chrysalis . . . . .	15
3.3.7	Result . . . . .	15
3.4	Ethereum overhaul . . . . .	16
3.4.1	Problem statement . . . . .	16

3.4.2	Solution . . . . .	16
3.4.3	Implementation . . . . .	17
3.4.4	Result . . . . .	18
3.5	Summary of Improvements . . . . .	19
<b>4</b>	<b>Open Issues</b>	<b>21</b>
4.1	Integrating Hyperledger into Chrysalis . . . . .	21
4.2	Improving on the Process Model . . . . .	21
<b>5</b>	<b>Caterpillar</b>	<b>23</b>
5.1	Caterpillar Section 1 . . . . .	23
<b>6</b>	<b>Resources</b>	<b>25</b>
6.1	Configuration Files . . . . .	25
6.2	Setup Guides . . . . .	25
6.3	Useful Links . . . . .	25
6.4	Other Documentation . . . . .	25
<b>7</b>	<b>Conclusion</b>	<b>27</b>
7.1	Conclusion Section 1 . . . . .	27



# 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 Business Processes on Blockchain

## 1.2 Problem Statement

## 1.3 Results

## 1.4 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 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 modeled 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**

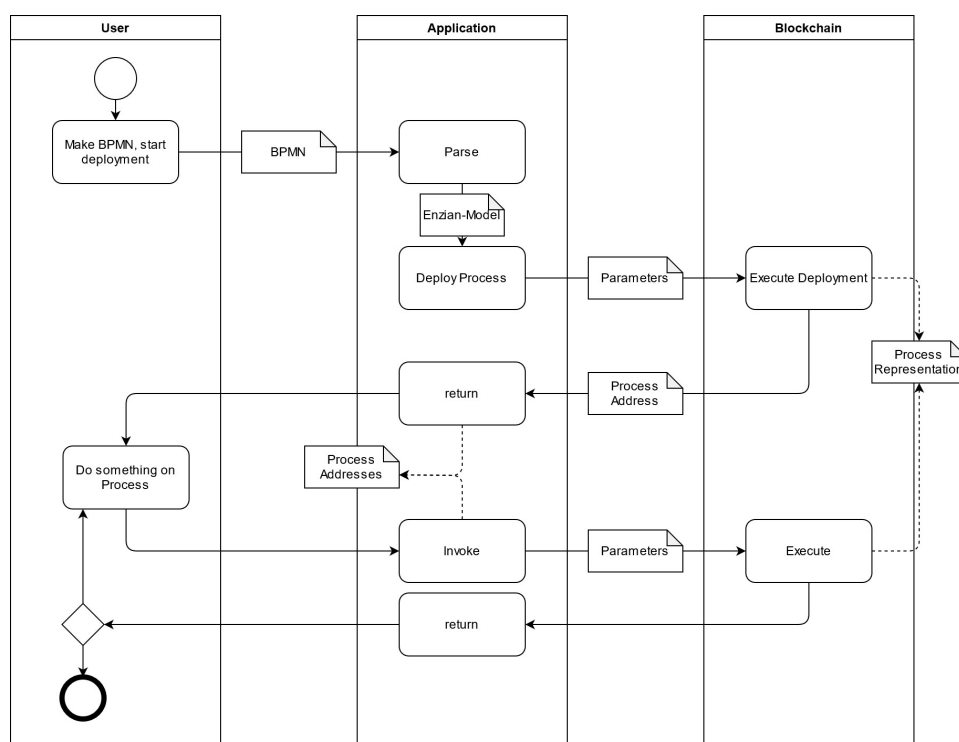
Finally, we will evaluate the success of our project and speak about the opportunities and limits it offers to those interested in deploying a decentralized process management engine.

## Architecture of Chrysalis

Since *Chrysalis* was handed to us as an already running prototype including a predefined layer structure and a way to interact with it in a browser, those designs were therefore a given. The Application is written entirely in JavaScript code, with the code deployed to the Blockchains being an exception sometimes. The prototype version was able to run on an *Ethereum*-Blockchain-Network, demonstrating it's functionality. In this chapter, we elaborate on the general structure of *Chrysalis*, omitting technical details and focusing on how the user and the components interact.

## 2.1 Intended Usage

From the user's point of view, three interactions with the system are generally intended, as they're described below. The configuration aside, the other two, encapsulating BPM, are shown in figure 2.1.



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

## Configuration

TODO config

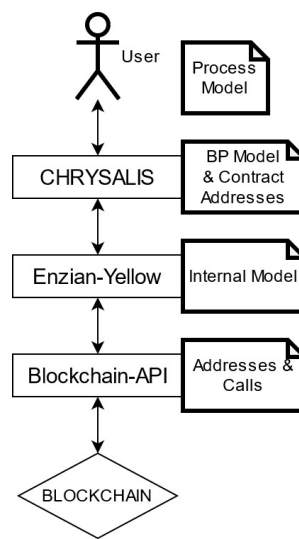
## Process Deployment

TODO deployment

## Process Execution

TODO execution

## 2.2 Components and their Interactions



**Fig. 2.2:** Layer structure of the original *Chrysalis* System.

The name-giving component, *Chrysalis*, is a front-end application written in *React* (for disambiguation, we will refer to the component as 'Chrysalis' and the entire system as 'project'). While it serves as a window for the user to interact with, it also handles the storage of any relevant data, like the addresses of deployed processes and authentication credentials of the user. *Chrysalis*, like any React-app, deploys a packaged version of itself and its dependencies into the user's browser, to be executed there. As for business process interactions, it instantiates an *Enzian-Yellow* object with the fitting configuration and delegates all commands to it.

*Enzian-Yellow*, being an installed dependency of *Chrysalis*, does the abstraction work between BPM actions and Blockchain operations. When instantiated and configured, it will in turn instantiate and configure a Blockchain-API to connect to the corresponding network node. *Enzian-Yellow* offers methods like creating and deploying processes and executing tasks, which are internally translated into the corresponding API invocations, and vice versa for the node's responses.

The *Blockchain-API* is the gateway from a local program to interact with a specified Blockchain network node. When instantiated, it establishes a connection to the specified node, handles authorization work and abstracts the networking away, so that the Blockchain's data and functions may be used as if they were local to the code.

The *Blockchain Network Node*, depicted in figure 2.2 as 'Blockchain', usually acts like a server that has a local copy of the Blockchain. It is fully synchronous with the network and any operation on the chain (i.e., addition of blocks) will be mirrored on every node in the network. In our case this means that every BPM action will be synchronized for every network participant that way.

## 2.3 Modeling of Processes

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

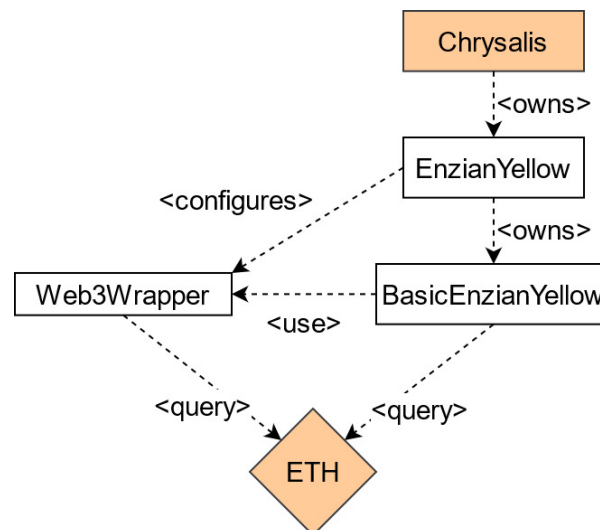


# Improvements

TODO introduce chapter

## 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

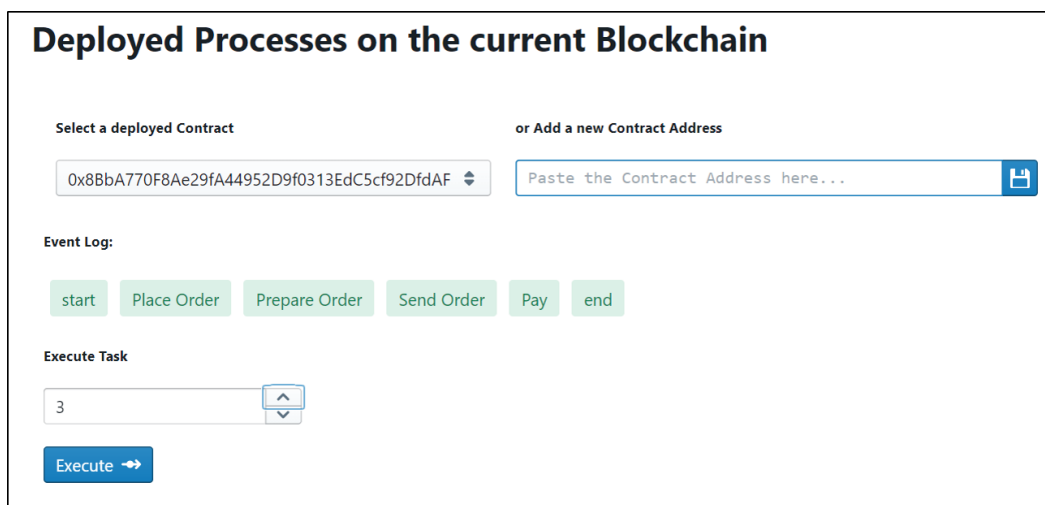
## 3.2 Persistence layer

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.



**Deployed Processes on the current Blockchain**

Select a deployed Contract: 0x8BbA770F8Ae29fA44952D9f0313EdC5cf92DfdAF

or Add a new Contract Address: Paste the Contract Address here...

Event Log:

start Place Order Prepare Order Send Order Pay end

Execute Task: 3

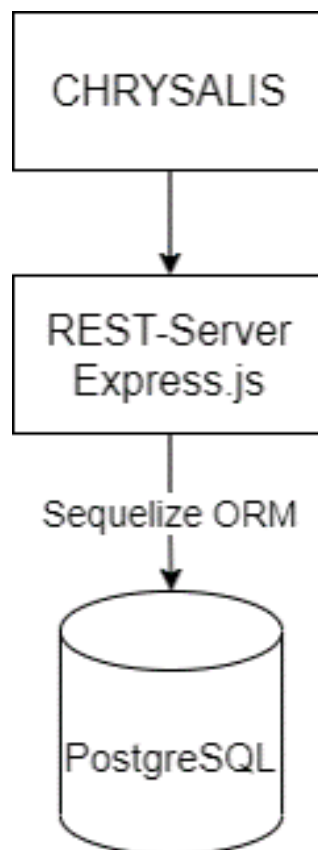
Execute

**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.

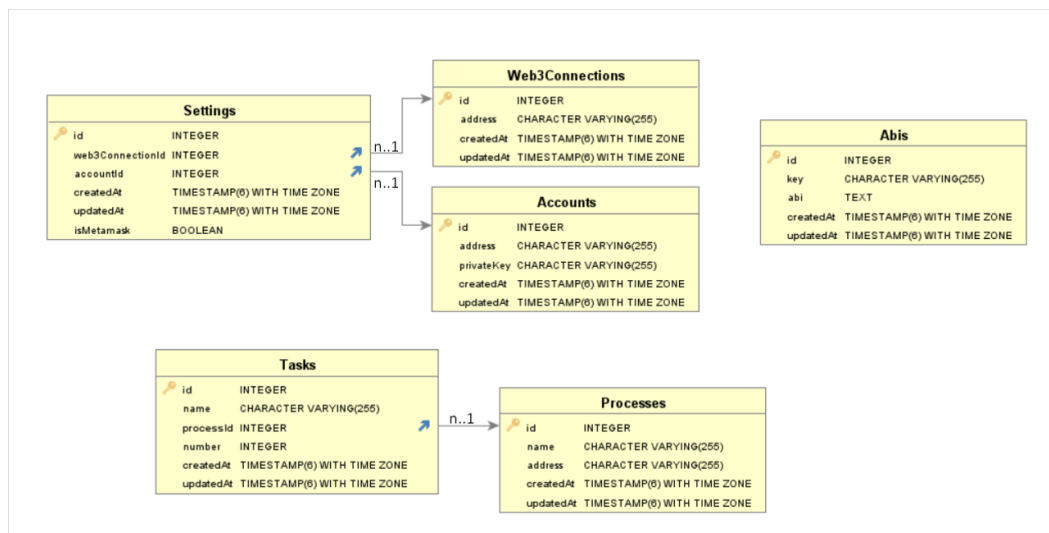


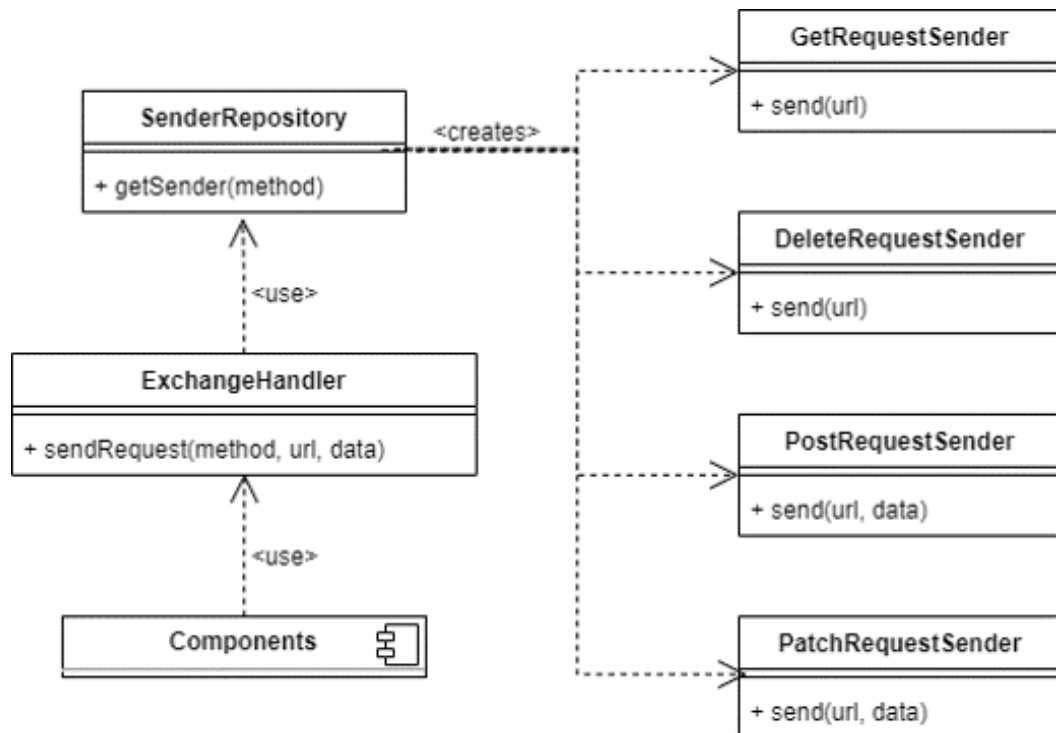
Fig. 3.4: Database schema

### 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, but 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 exchange 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

With the interfaces needed to easily expand the system being in place, we deemed it a good next step to add another Blockchain protocol. Specifically, we chose *Hyperledger*. The application was successfully expanded with all necessary components built and tested, although the front-end still has problems trying to pack this new addition and sending it to the user's browser. This issue is elaborated in section 4.1. This chapter will focus on firstly giving a broad overview of Hyperledger in section 3.3.1 and then following up with detailing the implemented expansion in the later sections.

### 3.3.1 Hyperledger as a Ledger Protocol

To explain how the Systems of *Hyperledger Fabric* work and what special features they offer, it makes sense to offer a comparison to the *Ethereum* Blockchain - feature by feature. *Hyperledger* wants to set itself apart by being a business-oriented information transfer protocol (hence, a shared ledger). Values, like tokens (e.g., *Bitcoin*, *Ethereum*), are not a fundamental part of it.

#### Authorization

If configured, a Hyperledger Network will be split into groups, called organisations. Each organisation has a (potentially distinct) *Certificate Authority (CA)*, through which membership in an organization is validated for every peer and account. Therefore, if one wants to interact with the network by for example invoking a smart contract, they must first be authenticated by the CA. The CA even allows for group roles, so not every query to the network is accessible to everyone. These group roles were not further regarded in this task, however.

### **Block Sealing**

Block sealing, the process of adding blocks to the Blockchain and therefore changing its active state and log, is an essential part of Blockchain technology. In more known protocols like *Bitcoin* or (at least at the time of writing) *Ethereum*, the content of the next block is decided by the node that first managed to solve a cryptographic puzzle - the solving of this puzzle being called *mining*. This, however, means that the time of sealing a block is somewhat random - and a bidding process is necessary to convince the block sealer of writing one's data.

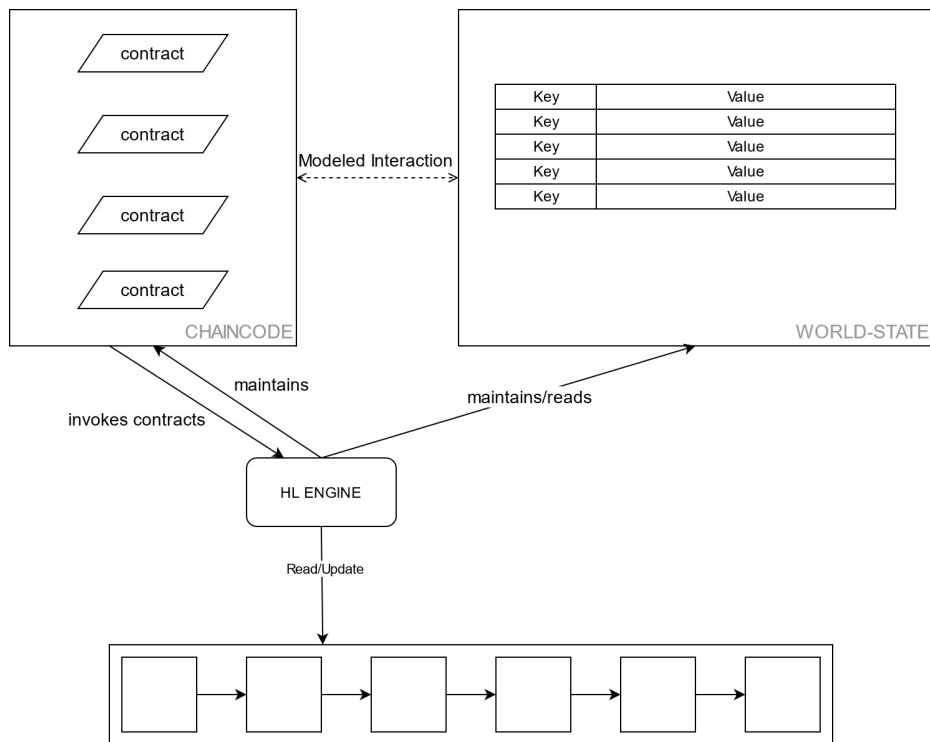
Hyperledger circumvents this competition-based method and instead introduces an *orderer* node, which generally tries to apply block-additions in a first-in-first-out fashion (and accounts for concurrency issues). This means a more reliable way of querying the blockchain, since all actions are executed as fast as resources allow, and also because competing peers are treated equally instead of based on their bid. Additionally, from a security standpoint, a network can not as easily be 'poisoned' (by holding such a large amount of tokens or mining power that the holder can essentially decide which transactions to incorporate), since the finances and mining capacities of peers are disregarded completely.

### **Abstraction: World State & Chaincode**

As depicted in figure 3.7 by an 'engine' component (although this is a big simplification), Hyperledger has functions in place to display the underlying Blockchain's contents in a more abstract way, so that the user or developer doesn't need to bother with physical addresses, but may instead find them in a more human-readable format.

The *World State* Container is a synchronous representation of all data present in Hyperledger's underlying Blockchain, meaning that the data points present are always a representation their most recent update. Additionally, the World State is arranged like a dictionary, so that every structure inside is reachable by a key, defined by the user itself. This way, the programmer doesn't have to worry about physical addresses of the data.

In the *Chaincode* containers one can find the Smart Contract objects one would also find in other Blockchain application. However, these contracts are not stateful and therefore do not contain any data besides the location of the World State, where the



**Fig. 3.7:** Sketch of *Hyperledger's* abstraction mechanism.

data may be contained. In comparison, an *Ethereum*-based contract may have private data and would therefore be stateful. Additionally, Chaincodes are also invocable via name, specifically by a combination of the parent contract name and the function that is to be executed.

### 3.3.2 Required Components

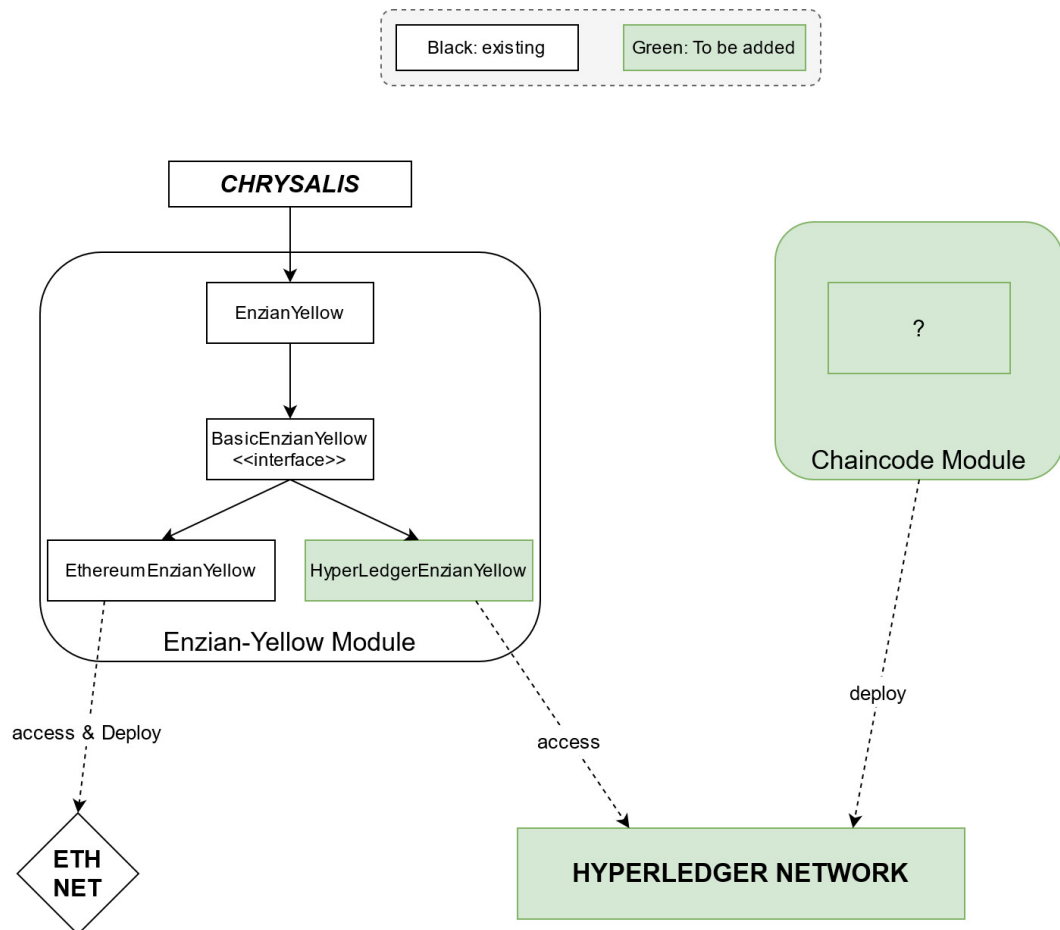
TODO explain what new components will appear and how they weave into EY

### 3.3.3 Test-Network

TODO as in pres

### 3.3.4 Representation of Processes

TODO as in pres



**Fig. 3.8:** Planned and implemented module structure after adding the *Hyperledger* protocol

### 3.3.5 Process Deployment and Execution

TODO as in pres

### 3.3.6 Integration into Chrysalis

TODO explain API a bit, otherwise as in pres

### 3.3.7 Result

TODO mention issue

## 3.4 Ethereum overhaul

In this section we will discuss improvements regarding the smart contract structure of the project. Smart contracts are used for the deployment and execution of process instances within the system. They contain structures and methods for creation and handling of tasks and control flow decisions. For each process instance one new instance of the BasicEnzian smart contract is deployed.

### 3.4.1 Problem statement

In its initial state, the project kept all of its on-chain functionality in one single smart contract. There was no separation of concerns, a lot of the universal, non instance-specific functionality was left in, which lead to not only poor scalability and reusability of the smart contract code in the future, but also to significant performance costs, which, on the ethereum network, are measured in a unit called "gas". Gas consumption defines how much computational power the network needs to execute a certain operation, which then determines how much currency should be transferred from the account of a node requesting to execute this operation, to the account of the owner of the node executing it. So in case of Ethereum, performance costs literally translate to money spent, that is why the main goal of the optimization was to minimize them.

### 3.4.2 Solution

To achieve this, we had to optimize the most expensive operation in our system - smart contract deployment, since it happens for every new process instance in the system. Also the issue of reusability of components becomes more crucial, because as the project will become progressively more complex in the future, the on-chain functionality will do so as well, so we have to split the smart contract code into elementary parts, which would allow to implement new features by adding and deploying new modules, using existing ones, instead of rewriting and redeploying everything, which would also decrease performance costs during future development.

As an implementation model for this task clean architecture was chosen, since this model is designed to increase reusability of application components and overall scalability of the application. The main principle of this model is that all the modules are divided into three layers - domain layer, application layer and presentation layer. Domain layer defines structures and entities used in the application, basic objects carrying its state. Application layer, or middle layer, contains business logic of the



application, the modules responsible for its behaviour, dealing with the object from the domain layer. Presentation layer provides entry points for external systems and users, that call functions and methods of the middle layer. The main requirement is that modules can only interact either with the modules from the same layer or the inner layer modules.

### 3.4.3 Implementation

On the domain model layer, the libraries containing structs defining the objects of a process model along with the process state were implemented. These libraries were TaskEntities, DecisionEntities and ProcessEntities.

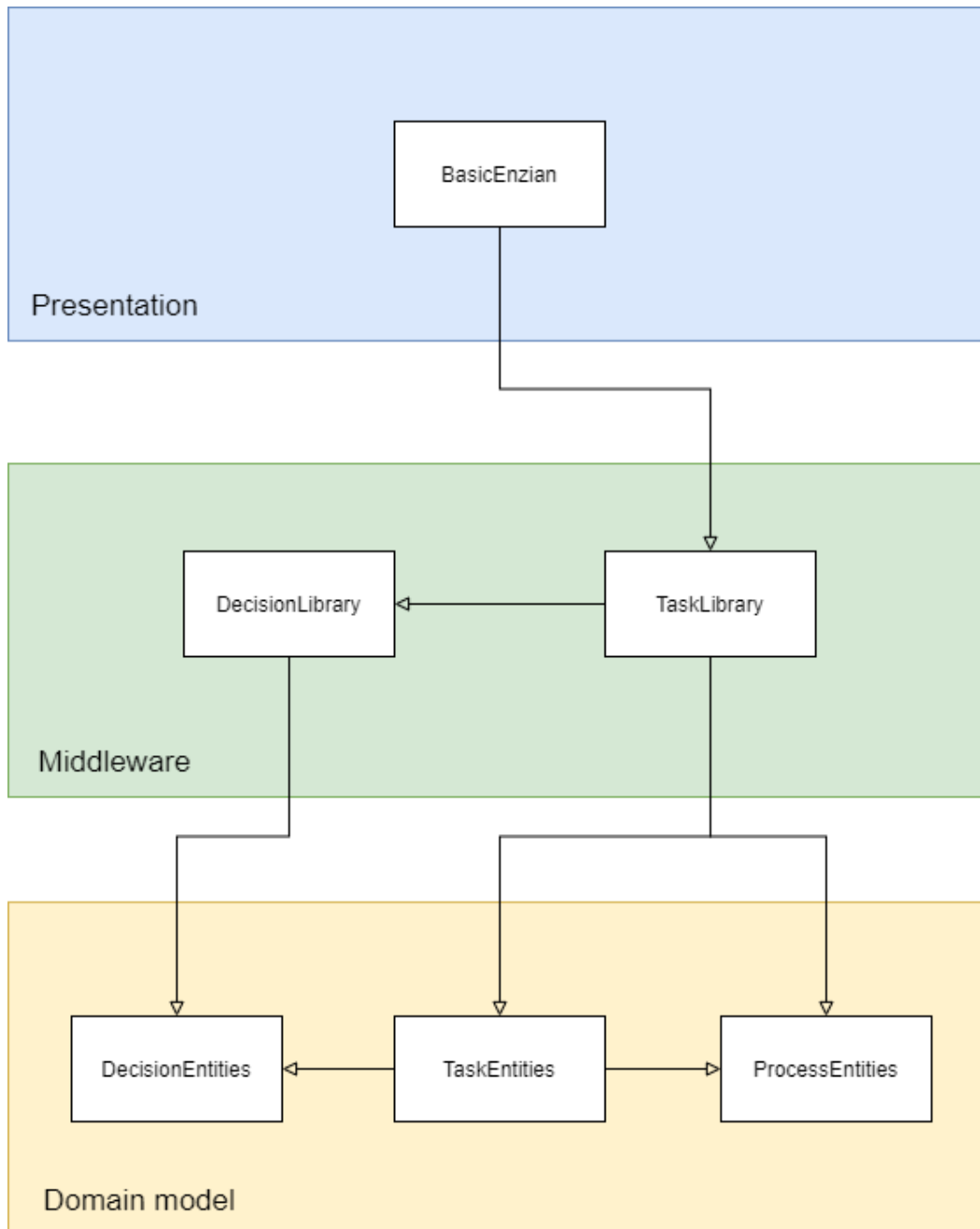
TaskEntities library define the structure of a process task, which contains information on Tasks name and id within the process instance, its completion state, as well as the set of requirements, that have to be fulfilled for this task to be up for execution. It also contains a reference to a Decision object, which is described in the DecisionEntities library.

DecisionEntities library defines structures and enumerators needed for evaluating control flow requirements in the gateways of a process model. The decision structure itself contains information on the gateway type, variables compared as well as the comparison operator for those variables. The enumerators containing gateway types and operators are also defined in this library.

ProcessEntities library defines the structure containing information on the process structure and state. It contains the process' tasks, its integer variables and string variables.

On the application layer there are two libraries: TaskLibrary and DecisionLibrary. TaskLibrary implements methods creating tasks on the process initialization, as well as methods handling execution of tasks including evaluation of task requirements. To evaluate gateway conditions it calls the DecisionLibrary, which implements appropriate functionality.

The presentation layer consists of the BasicEnzian smart contract, which provides entry points for the EnzianYellow library to interact with, as well as keeps the state of the process instance and its event log.



**Fig. 3.9:** Structure of the ethereum smart contracts

### 3.4.4 Result

As an outcome of the smart contract overhaul, the structure of the smart contract code became more scalable and flexible, it was divided into small modules which can be reused in the new contracts developed further down the line. Other than

that, the cost of process instance deployment was lowered from 4300000 gas to 3600000 gas, and overall the code became much cleaner, which could be seen in the following listing, which showcases the same function handling task execution. After the refactoring most of the method's logic is encapsulated in the TaskLibrary, which is called, and then the results from the library function are used to update process state.

**Listing 3.1:** Task execution after the optimization

```
function completing(uint taskId) public returns (bool success){
    require(!processState.tasks[taskId].completed, "DO NOT REPEAT
        TASKS!!!");
    TaskEntities.Task memory thetask = processState.tasks[taskId
    ];
    uint endBoss;
    (success, endBoss) = TaskLibrary.completeTask(thetask,
        processState);

    if(thetask.decision.exists) {
        processState.enabled[endBoss] = success;

        //LOCKING
        for (uint i = 0; i < thetask.competitors.length; i++) {
            processState.tasks[(thetask.competitors[i])].
                completed = success;
        }

    }
    processState.tasks[taskId].completed = success;
    emit TaskCompleted(success);

    if (success) {
        debugStringeventLog.push(thetask.activity);
        theRealEventLog.push(taskId);
    }

    return success;
}
```

## 3.5 Summary of Improvements

TODO section even necessary?



## Open Issues

TODO this is far from done even if we succeeded at our tasks

TODO this will be some hints what to do next

### 4.1 Integrating Hyperledger into Chrysalis

TODO explain why chrysalis doesn't eat enzian-yellow any more

TODO explain that deploying it is a bad idea anyway

TODO Task: put ey into a container and make it a server

TODO explain: good to have it close to the blockchain anyway, less networking issues

### 4.2 Improving on the Process Model

TODO our work was very architectural, now that thats done it would be perfect time to expand functionality



## Caterpillar

### 5.1 Caterpillar Section 1





## Resources

6.1 [Configuration Files](#)

6.2 [Setup Guides](#)

6.3 [Useful Links](#)

6.4 [Other Documentation](#)



## Conclusion

### 7.1 Conclusion Section 1



## List of Figures

2.1	Layer structure of the original <i>Chrysalis</i> System. . . . .	3
2.2	Layer structure of the original <i>Chrysalis</i> System. . . . .	4
3.1	Component structure of the original <i>Enzian-Yellow</i> Repository. The components marked in orange are not part of <i>Enzian-Yellow</i> , but interact with it. . . . .	7
3.2	Process execution page before implementation of the persistence layer	8
3.3	Persistence layer architecture . . . . .	9
3.4	Database schema . . . . .	10
3.5	Database exchange module . . . . .	11
3.6	Process execution page after the improvements . . . . .	12
3.7	Sketch of <i>Hyperledger's</i> abstraction mechanism. . . . .	14
3.8	Planned and implemented module structure after adding the <i>Hyperledger</i> protocol . . . . .	15
3.9	Structure of the ethereum smart contracts . . . . .	18



## List of Tables





# Declaration

Hiermit erklären wir, dass wir den vorliegenden Abschlussbericht selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen des Abschlussberichtes, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

*Bayreuth, August 6, 2021*

---

Philipp Scholz

---

TODO 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. Alternatively delete this and put your signature under the other declaration.

*Bayreuth, August 6, 2021*

---

Anatoly Obukhov

