University of Zurich[UZH]

# Design and Implementation of a Blockchain Intent Management System

*Sandro Padovan*
*Zürich, Switzerland*
*Student ID: 17-721-291*

Supervisor: Eder John Scheid, Muriel Figueredo Franco
Date of Submission: November 1, 2020

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

ifi

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

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

# Abstract


### Deutsch

Das Konzept von Intents wurde 2015 erstmals im Kontext von Intent-based Networking angewendet, jedoch bald auch in anderen Kontexten wie beispielsweise in Cloud Management. Ein Intent kann als eine Policy mit hohem Abstraktionsgrad definiert werden, bei der ein Ziel definiert wird, jedoch nicht spezifiziert wird, wie dieses Ziel erreicht werden soll. Diese Bachelorarbeit wendet Intent-based Management auf die Blockchain-Auswahl an. In den letzten Jahren sind zahlreiche verschiedene Blockchains entstanden, wodurch die Auswahl für einen bestimmten Anwendungsfall immer komplexer wurde. Um den Prozess der Auswahl einer geeigneten Blockchain zu vereinfachen, wird ein Blockchain Intent Management System (BIMS) entworfen, implementiert und evaluiert. Das vorgestellte System integriert das Intent Refinement Toolkit (IRTK) für die Übersetzung von Intents in Policies sowie ein Policy-based Blockchain Selection Framework. Zur intuitiven Eingabe eines Intents wird eine kontrollierte natürliche Sprache verwendet, unterstützt durch Textvorschläge und Autovervollständigung.

### English

With the concept of intents first being introduced in intent-based networking in 2015, the concept soon was applied in other contexts such as cloud management. Intents can be defined as abstract, high-level policies which describe "what" the goal is but not "how" a system should achieve this goal. This bachelor thesis applies intent-based management to blockchain selection. Over the last years, numerous different blockchains emerged, making the selection for a specific use-case more and more complex. To simplify the process of selecting a suitable blockchain based on high-level requirements (*i.e.* intents), a Blockchain Intent Management System (BIMS) is designed, implemented and evaluated. The presented system integrates the Intent Refinement Toolkit (IRTK) for the translation into lower-level policies, as well as a policy-based blockchain selection framework. For an intuitive specification of an intent, a controlled natural language is used, supported by text suggestions and auto-completion.

# Acknowledgments

I would like to thank my supervisor Eder J. Scheid for his great long-distance support and flexibility, especially in these tumultuous times during COVID-19.

# Contents

**Abstract** **i**

**Acknowledgments** **iii**

**1 Introduction and Motivation** **1**

1.1 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**2 Background** **3**

2.1 Policy-Based Management (PBM) . . . . . . . . . . . . . . . . . . . 3

2.1.1 Policy-Based Management Architecture . . . . . . . . . . . . . 4

2.2 Intent-Based Management . . . . . . . . . . . . . . . . . . . . . . . 5

2.2.1 Intent-Based Networking (IBN) . . . . . . . . . . . . . . . . . 6

2.2.2 Intent versus Policy . . . . . . . . . . . . . . . . . . . . . . . 6

2.2.3 Intent Lifecycle . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.3 Blockchain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.3.1 Blockchain Characteristics . . . . . . . . . . . . . . . . . . . . 8

2.4 Intent Refinement Toolkit (IRTK) . . . . . . . . . . . . . . . . . . . 9

2.5 Policy-based Blockchain Agnostic Framework - PleBeuS . . . . . . . . . . . 10

**3 Related Work** **13**

3.1 Network Management . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.2 Cloud Management . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3.3 Blockchain Selection . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

**4 Blockchain Intent Management System** **17**

4.1 BIMS Design . . . 17

4.1.1 Database . . . 19

4.1.2 REST API . . . 20

4.1.3 User Interface . . . 20

4.2 Implementation . . . 22

4.2.1 Technologies . . . 23

4.2.2 Intent Auto-Completion . . . 24

4.2.3 IRTK Extension . . . 27

4.2.4 PleBeuS Integration . . . 28

**5 Evaluation and Discussion** **31**

5.1 Software Testing . . . 31

5.2 Case Studies . . . 32

5.3 Discussion . . . 37

**6 Summary and Future Work** **39**

6.1 Future Work . . . 39

**Bibliography** **41**

**Abbreviations** **45**

**List of Figures** **46**

**List of Tables** **47**

**A Installation Guidelines** **51**

A.1 Code Structure . . . 51

A.2 Setup . . . 52

A.3 Troubleshooting . . . 54

**B Contents of the CD** **57**

# Chapter 1

# Introduction and Motivation

The concept of intents was first introduced in 2015 within the context of autonomic networking [1], leading to the creation of Intent-Based Networking (IBN). In the IBN context, intents can be viewed as high-level abstract policies and define "what" and not "how" a networking system should achieve the defined goal [2]. However, there is still not a consensus on the level of abstraction and the definition of intents [3, 4, 5] from the network management community. Thus, this concept is being actively discussed and presents an interesting research topic.

Further, the concept of intents is not restricted to network management, being applied to several areas and systems, from cloud management [6, 7] to blockchain selection [8]. The employment of intents helps to reduce the manual interaction and the technical knowledge from users of the systems to achieve a particular task, for example, selecting the most suitable blockchain based on high-level requirements, which is the topic of this thesis. Intents employed in the thesis follow the "for *clientX*, *clientY* and *clientZ* select the *cheapest public*, *stable* and *popular* blockchain *except Bitcoin* and *Ethereum* with *splitting*, *redundancy* and *encryption* until the *daily* costs reach *CHF 10*" format, where the italic words are automatically extracted from the sentence.

Thus, this thesis presents a Blockchain Intent Management System called `BIMS`. `BIMS` implements functions to *(i)* allow users to input (*i.e.,* write) their intents regarding blockchain selection, *(ii)* manage users (*e.g.,* authentication), *(iii)* provide feedback to users (*e.g.,* visual notifications), and *(iv)* manage intents with basic Create, Retrieve, Update and Delete (CRUD) functions. Furthermore, by being integrated with the Intent Refinement Toolkit (IRTK) [9] and the Policy-based Blockchain Selection framework (PleBeuS) [10], `BIMS` is able to *(a)* automatically refine an intent and provide text auto-completion, and *(b)* create the refined low-level policy directly in *PleBeuS* using the exposed REST Application Programming Interface (API). `BIMS` was evaluated using user stories to verify its functionality and feasibility.

## 1.1 Thesis Outline

This thesis is outlined as follows. In Chapter 2, a theoretical background is established, covering policy-based management, intent-based management, blockchains, as well as preliminary work which this thesis is based on. Chapter 3 discusses different approaches of intent-based management with a focus on the way how intents are specified in different contexts. In Chapter 4, the design, architecture and implementation of the Blockchain Intent Management System are presented in detail. Further, the selected technologies are discussed and explained. An evaluation of the developed prototype is provided in Chapter 5. To conclude, a summary and an outlook for future work is given in Chapter 6.

# Chapter 2

# Background

This chapter covers the fundamental theoretical basis of this thesis. In Section 2.1, the concept of policy-based management is explained. Following in Section 2.2, the concept of intent-based management is explained, first in the context of intent-based networking, then generally in other contexts. Afterwards, in Section 2.3, the concept of blockchains is covered. Next, in Section 2.4, the Intent Refinement Toolkit is explained and finally in Section 2.5, the Policy-based Blockchain Agnostic Framework *PleBeuS* is introduced.

## 2.1 Policy-Based Management (PBM)

As [11] documents, the first use of policies goes back to 1966 in the context of security policies. In order to regulate access control, security policies were used to define (high-level) rules. By 1985 policies found its way into network management [11]. As networks had become increasingly complex and devices more heterogeneous, policy-based management was successfully employed to simplify and automate network administration [12].

Later, in 1999, the Internet Engineering Task Force (IETF) and the Distributed Management Task Force (DMTF) developed a standardized policy framework. Further, they defined a policy as a "set of rules to administer, manage, and control access to network resources" [13]. A policy consists of a tuple of Events, Conditions and Actions (ECA) [14]. Thus, on an event, if the condition of a policy is met, the respective action or set of actions is executed [15].

Policies vary in terms of abstraction, terminology, role of the policy author and level of detail. For instance, both a business rule and a device configuration can be represented by a policy, although their level of abstraction are different. In order to ensure consistency, a policy continuum was proposed [14]. The policy continuum is a modelling tool with five different views each representing a level optimized for a specific use and level of abstraction, as shown in Figure 2.1. Once created, a policy is then translated and refined to lower levels of the policy continuum. This refinement process can influence other policies on lower levels, creating one or more policies. According to the different levels, a policy may be created on any level and can, but does not have to affect another level [14].

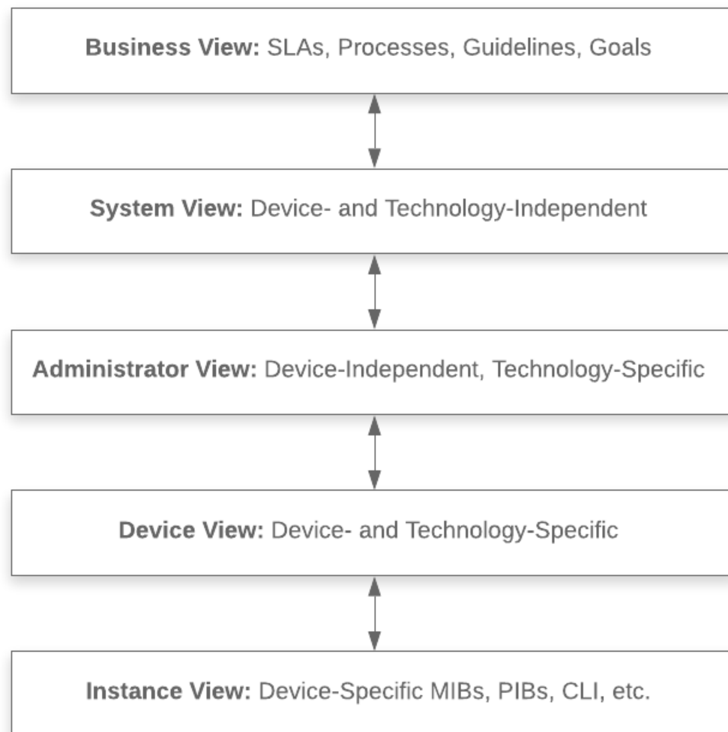


Figure 2.1: Policy continuum

### 2.1.1 Policy-Based Management Architecture

In this subsection, the policy management architecture proposed by the IETF/DMTF is explained. Figure 2.2 depicts the architecture in a graphical way, which consists of four components [12, 13]:

- **Policy Management Tool (PMT):** Tool which an administrator uses to define and manage policies.
- **Policy Decision Point (PDP):** Point which evaluates a policy's condition and communicates this decision to the PEP.
- **Policy Enforcement Point (PEP):** Point which applies and executes different policies. In a network context, the PEP can be viewed as a device (*e.g.,* a router).
- **Policy Repository:** Stores policies managed by the PMT.

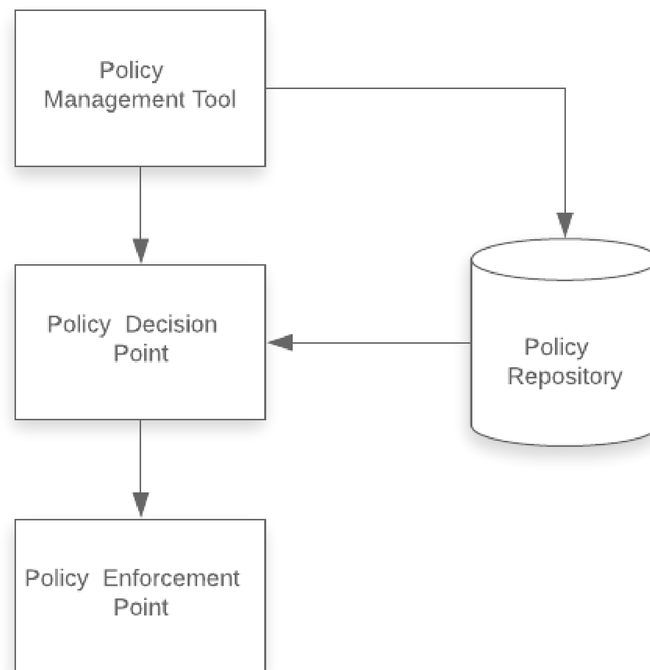


Figure 2.2: Policy-based management architecture

With this architecture, two goals are accomplished: *(a)* centralization and *(b)* business-level abstraction. The PMT is the central point where configurations are defined as opposed to configuring each individual device. With this centralization, manual workload can be reduced especially in large networks with many devices, as one definition of a policy makes the configuration of many individual devices obsolete. By defining policies with less specific details of technology, much of the complexity can be abstracted [12]. For instance, the author of a policy defined in the top layer of the policy continuum is concerned about Service Level Agreement (SLA) information, but does not need to know the specific technologies used [14]. Thus, it simplifies policy and network administrator's tasks [12].

## 2.2 Intent-Based Management

In 2001, autonomic computing systems were described with the idea of self-managing with high-level policies representing goals and restrictions of a system [16]. Since then, the idea of self-managing and autonomic systems was applied, first in networking as Intent-Based Networking (IBN), and later in various other contexts such as cloud service management [1, 6]. Many of the fundamental concepts first used in IBN were applied in other contexts in intent-based management.

### 2.2.1 Intent-Based Networking (IBN)

In autonomic networks, the network is self-managing, meaning it is self-configuring, self-protecting, self-healing and self-optimizing [1], applying the concepts first described in autonomic computing systems [16]. Although the network is self-managing, a central entity can influence its management with the definition of an intent. Self-configuration means that the configuration happens without manual effort (*i.e.,* without using user interaction) or a management system, but by the network itself based on self-knowledge, discovery and intents [1].

An intent in autonomic networks is defined as "*An abstract, high level policy used to operate the network*" [1]. According to this definition, an intent can thus be viewed as a type of policy. The question of differences between policy and intent is discussed in-depth in Section 2.2.2. An intent contains abstract information; thus, not concerning a specific network component or configuration. In effect, an intent specifies "what" should be achieved but not "how" to accomplish it [3]. Not just the definition of an intent is at a very abstract level, but also the reporting and feedback does not contain specific low-level details. Further, information about the effectiveness of a specified intent must be reported to an administrator, but still on a high abstraction level [1].

There are several operating principles of intent-based systems. An intent-based system must have a **Single Source of Truth (SSoT)**, *i.e.,* the set of validated intents. This allows for comparing the intended state and the actual state of the system. Moreover, an intent-based system should be implemented in a **one-touch** fashion, meaning that a user only specifies an intent, the rest is executed by the system. Intent-based systems must be **autonomous** but give an **oversight** to the user. This means that the system must be able to autonomously take decisions without intervention of the user in order to fulfil the intent.

However, reporting of relevant information to the user at the same time is crucial in order to give an oversight. In contrast to ECA policy rules, in an intent-based system, the user only specifies the goal but not what measures must be taken if a specific condition is met. Hence, it is a **learning** system as the decisions are taken by the system, not the human. In addition, intent-based systems are continuously changing, creating a need for continuous learning. Mapping the users goals and expectations to actions performed by the system needs **explainability** to bridge the semantic gap between the intent and the actions performed [3].

### 2.2.2 Intent versus Policy

The distinction between intent and policy is not always clear, sometimes the two terms are used synonymously. Both intent and policy abstract the details (for example device-specifics) and have the goal of simplifying the managing of a complex system; however, there are differences. A policy is typically a ECA-rule and it does not specify a desired outcome. In contrast, an intent specifies a desired outcome and there is no need of specifying specific events, conditions or actions. An intent is at a higher level of abstraction

and is declarative, it only specifies the goal, not how to reach it [3]. In [1], an intent is defined as a high-level policy, so intents could be seen as a subset of policies, located in the top layer of the policy continuum [14]. As shown in Figure 2.1, the *Business View* includes goals, under which intents could be incorporated.

### 2.2.3 Intent Lifecycle

In [5], a distinction is made between *transient* and *persistent* intents. *Transient* intents have no lifecycle management, the intent is finished once its operation is successfully carried out. In contrast, a *persistent* intent is kept active by the system until it is deactivated or removed. Thus, *persistent* intents have lifecycle management. In [3], a model of an intent lifecycle is proposed, shown in Figure 2.3. It makes a distinction between three spaces: the User Space, the Translation / Intent-based System (IBS) Space and the Network Operations Space. Further, there is a distinction between Fulfillment and Assurance. Fulfillment comprises the functions from the definition by a user to the enforcement in the system. Assurance on the other hand contains the functions to make sure that the system complies with the specified intent and reporting relevant information to the user.

As one can see in Figure 2.3, there are two cycles in the intent lifecycle. The inner loop does not involve the user space. It is therefore completely automated and does not need any human interaction. The outer cycle involves the user where intents are created, updated and reports with abstracted information are received [3].

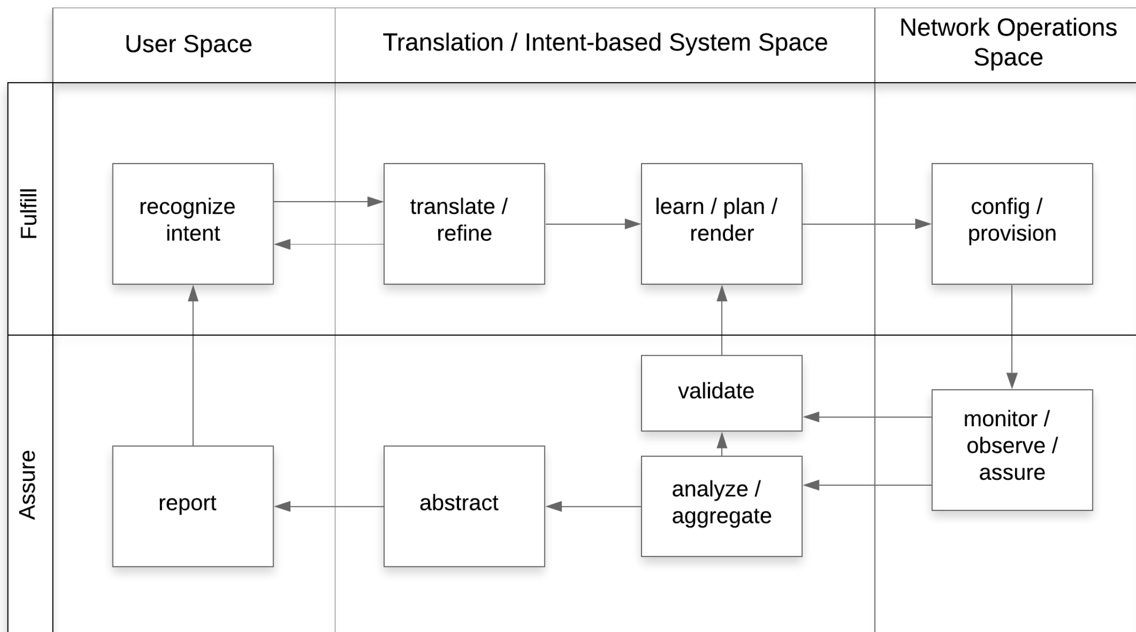


Figure 2.3: Intent lifecycle

In an IBS, five modules are proposed: Intent Management, Translation, Decision, Verification and Intent Repository Module. The Intent Management Module receives the

intents from the user. If it is invalid, it is fed back to the user, however if it is valid, the intent is forwarded to the Translation Module. The Translation Module translates and refines the intent into a lower-level policy or configuration. The Verification Module then assures that the configuration is able to be executed. After this, the Decision Module processes the data and decides whether the configuration is executed. The Verification Module also monitors the status of the system and detects if optimization is necessary, which the Decision Module would carry out. Finally, the Intent Repository Module is where intents and configurations are stored [4].

## 2.3 Blockchain

With Bitcoin, the concept of blockchains was first introduced as a Peer-to-Peer (P2P) electronic cash system [17]. Its main idea was to give a solution to the double-spending problem and not having to rely on a Trusted Third Party (TTP), such as a financial institution. A blockchain is an append-only, immutable ledger, stored in a distributed manner. It stores transactions, but unlike conventional databases, there is no central entity controlling the data and access rights. Every node stores a copy of the blockchain and transactions are broadcast to the network. These transactions are stored in blocks containing multiple transactions, with each block having a uniquely identifying hash value. Timestamps are used to form a chain of blocks, with each block also storing the hash of the previous block. In order for blocks to be added to the blockchain, a consensus mechanism is needed to guarantee that the majority of nodes accept the new block. All these concepts together form a technology applicable in various use cases and different variations of blockchains [18].

### 2.3.1 Blockchain Characteristics

There are different types of blockchains with different characteristics such as the deployment type, consensus mechanism, performance, and costs.

#### Deployment Type

Usually, blockchains can be categorized in three different deployment types: *(a)* public, *(b)* private or *(c)* consortium blockchain. A public blockchain, also known as a permissionless blockchain [19], has no restriction on access of read and write permissions. As every node has a synchronized copy of the blockchain, it is immutable and completely decentralized. The blockchain is open for everyone to join, making the consensus mechanism crucial. On the contrary, in private blockchains (also known as permissioned blockchains [19]) access is regulated and controlled by an organization or a set of rules. Also, writing permission might be restricted for some nodes in the network. The entity controlling access to the blockchain makes private blockchains a centralized system. A consortium blockchain is a private blockchain with a set of special validator nodes which

control access and mining of the blockchain. Consortium blockchains are partially centralized. Thus, the deployment type differences need to be considered when planning for a specific use case [18].

### Consensus Mechanism

The consensus mechanism is a crucial part of a blockchain enabling users to have trust that the data in the blockchain network is correct (given the protocol); however, there are several different approaches to reach consensus. **Proof-of-Work (PoW)** was the first approach introduced by Bitcoin. In a PoW-based approach, nodes solve a computationally expensive cryptographic puzzle to find the block hash that is below a certain threshold in a process called mining. In case of forks in the blockchain, the longest chain (*i.e.,* the chain with the highest accumulated puzzle difficulty) is the one accepted as valid by the majority of nodes. As an incentive for users to support the network, a reward is given to the miners. As long as honest nodes collectively have the most computing power, the system can be trusted [17]. The drawback of PoW is its high energy consumption [18].

To tackle its high energy consumption in the process of mining, **Proof-of-Stake (PoS)** was proposed as a solution by the cryptocurrency Peercoin [20]. In PoS, block creators are randomly chosen with the wealth (*i.e.,* stake) of a node affecting the probability of being chosen. Similar to PoW, the nodes which create new blocks also get an incentive, but if the new block is not included in the existing chain, there is a penalty. This is to solve the so called nothing-at-stake problem. If there are multiple chains competing, new blocks would be created on every chain, not just on the longest since there are no additional costs and the expected value is increased. Without the penalty, the blockchain may never reach consensus [21].

Over time, a variety of other approaches than those mentioned or variations of those have come up to solve this problem, such as delegated-Proof-of-Stake (dPoS) [22], Proof-of-Authority (PoA) [23] or Proof-of-Elapsed-Time (PoET) [24].

## 2.4 Intent Refinement Toolkit (IRTK)

In this section, the Intent Refinement Toolkit (IRTK) is explained. The IRTK is a tool to refine and translate intents to low-level policies. For the intent specification, the authors of the IRTK use a Controlled Natural Language (CNL) based on English as the input language [9, 8]. First, the intent is parsed and validated. After this phase, the parsed intent is stored as a intermediary data structure with all the parsed options. After this phase, the parsed intent is translated in a set of low-level policies and again validated [9].

The intent specified in the input language is composed of different parameters, namely *Conditions, Selections, Filters* and *Modifiers*. *Conditions* include the intent parameters **Users, Timeframe, Cost Interval, Cost Currency** and **Cost Threshold**. The *Selection* strategies include **Profile** or a particular **Blockchain**. *Filters* include **Deployment Type, Transaction Rate, Transaction Costs, Maturity, Whitelist** and **Blacklist**.

Finally, *Modifiers* include **Redundancy, Encryption** and **Splitting**. Table 2.1 explains the intent parameters in more detail [9].

With the parameters listed in Table 2.1, an intent is structured as follows:

```
for <users> [ in the <timeframe> ] select ( the <profile>
[ <filters> ] blockchain [ ( from | except )  <blockchains> ]
| <blockchain> ) [ with <modifiers> ] ( until the <interval>
costs reach [ <currency> ] <threshold> | as default )
```

Architecturally, the IRTK is composed of several components. The Refiner component is the entry-point and acts as a wrapper for the parser and the translator. While parsing an intent, the Parser component relies on the Tokenizer component and on the Validator component. The Parser of the IRTK is implemented as a Deterministic Finite Automaton (DFA) with states according to the expected tokens of an intent. Once parsed, the intent is translated into a policy by the Translator component.

## 2.5 Policy-based Blockchain Agnostic Framework - PleBeuS

*PleBeuS* is a PBM framework used for blockchain selection. It lets users define policies as requirements for the selection process. With the policies as input, with filtering and selection algorithms, a blockchain is then selected. Additionally, *PleBeuS* connects to a blockchain interoperability API which lets users create transactions to the selected blockchain.

Architecturally, *PleBeuS* is implemented with a graphical user interface making requests to an API. Internally, the system is split into two components, the PMT and PDP (see Section 2.1.1). The PMT incorporates an API request handler and a database storing user, policy, blockchain and transaction data. The PDP includes a Policy Selector, Blockchain Selector and a Transaction Generator which connects to the blockchain interoperability API to execute the transactions [10].

The policies consist of different parameters [10]:

- *Public vs. Private*: Select a public or a private blockchain.
- *Blockchain Throughput*: Minimum amount of transactions per second (tps).
- *Data Size*: Minimum amount of bytes which a transaction must allow to include.
- *Turing-Completeness*: Select a blockchain which supports complex smart contracts.
- *Cost Thresholds*: Maximum cost for a specific interval.
- *Cost Interval*: Interval for the cost threshold. Supports *(a)* **daily**, *(b)* **weekly**, *(c)* **monthly**, *(d)* **yearly** and *(e)* **default**.

Table 2.1: Intent parameters of the IRTK

<table>
<tr><th>Parameter</th><th>Possible values</th><th>Description</th></tr>
<tr><td>users</td><td>set of users, separated by "," | "and".</td><td>One or more users, which defines his/her own independent policies.</td></tr>
<tr><td>user</td><td>string</td><td>A user is specified with a string of characters.</td></tr>
<tr><td>timeframe</td><td>day, night, morning, afternoon</td><td>Timeframe, in which a policy can be activated.</td></tr>
<tr><td>profile</td><td>cheapest | fastest</td><td>Determines how a blockchain is selected. The cheapest profile selects the blockchain with the lowest transaction cost, the fastest profile the blockchain with the highest transaction rate.</td></tr>
<tr><td>filters</td><td>set of filters, separated by "," | "and".</td><td>Multiple filter options.</td></tr>
<tr><td>filter</td><td>private | public | fast | cheap | stable | popular</td><td>Optional filter options to specify deployment type, transaction rate, transaction costs, maturity and a whitelist / blacklist to restrict the blockchain pool.</td></tr>
<tr><td>blockchains</td><td>set of blockchains, separated by "," | "and".</td><td>Multiple specific blockchains.</td></tr>
<tr><td>blockchain</td><td>Bitcoin | EOS | Ethereum | Hyperledger | IOTA | Multichain | Stellar</td><td>A particular blockchain to store transactions. Values depend on blockchains which are supported.</td></tr>
<tr><td>modifiers</td><td>set of modifiers, separated by "," | "and".</td><td>Multiple filter options.</td></tr>
<tr><td>modifier</td><td>encryption | redundancy | splitting</td><td>Optional modifier options including redundancy (additionally store data in conventional database), encryption (encrypt transaction data) and splitting (choose different blockchain for every transaction).</td></tr>
<tr><td>interval</td><td>daily | weekly | monthly | yearly</td><td>Defines at which rate the accumulated transaction costs are reset.</td></tr>
<tr><td>currency</td><td>CHF | EUR | USD</td><td>Currency of the cost threshold. Default is USD.</td></tr>
<tr><td>threshold</td><td>integer | real</td><td>Threshold for accumulated transaction costs. If costs exceed the threshold, the policy is deactivated. A threshold is required, except for default policy, which does not specify a threshold.</td></tr>
</table>

- *Fiat Currency*: Currency for threshold. Supports **USD**, **CHF** and **EUR**.
- *Cost Profile*: This parameter is used if multiple blockchains fulfil the criteria. Supports **economic** and **performance**.
- *Transaction Split*: Boolean whether split transactions are allowed.
- *Time Frame*: Time frame when a policy should be active.
- *Preferred Blockchains*: Select specific blockchains for the selection process.

# Chapter 3

# Related Work

Apart from intent-based networking, the concept of intents has been used in other contexts as well, for example cloud management [6, 7], datacenter solutions [25] or in case of this thesis, blockchain selection [9]. While most of IBS are applied in networking, the same concepts are used in other contexts. With intent-based management being applied in more and more different use cases, the variety of different implementations of systems also grows. In this chapter, different approaches of IBS are discussed in different contexts. Section 3.1 focuses on intent-based network management, Section 3.2 on its application in cloud management, Section 3.3 on blockchain selection. Finally in Section 3.4, the different approaches are discussed and compared.

## 3.1 Network Management

Being the origin of intent-based management, networking shows a great variety of uses of intents and different ways of specifying them. In [26], an intent is specified in the form of <*verb, object, modifiers, subject*> tuples. A *verb* is classified in three different categories: *(a)* Construct, *(b)* Transfer or *(c)* Regulate. Each of these categories holds a set of primitive verbs which are used in the intent specification, like *Discover, Advertize, Push, Pull, Block, Prioritize* or *Allocate*. An *object* identifies the entity (service, process or item) which is the objective of the verb. With the *modifiers*, the object is parameterized. Finally, the *subject* is an optional identifier of a linked service/process/item. After being formulated, the intents are compiled and refined in the network.

In [27], the authors propose a IBN solution called INSpIRE to refine and translate intents into a set of configurations. From an intent, INSpIRE determines the Virtual Network Functions (VNF) needed to fulfil the intent, chains the VNFs according to dependencies, and provides low-level information to network devices. Intent specification happens in a CNL, meaning that the natural language is constrained by restricting grammar and vocabulary.

Extending the refinement process of intents, the authors in [2] propose the use of Artificial Intelligence (AI) to translate intents. A chatbot interface is used for users to input their

intents in natural language, enabling the deployment in different scenarios and across multiple platforms. Using AI, the key aspects are extracted from the input. Further, the authors propose a structured intent definition language called *Nile* which resembles natural language. The intents are translated into a network definition program written in *Nile*, acting as an abstraction layer for other policy mechanisms. Finally, this program is compiled into a network policy.

## 3.2 Cloud Management

Cloud services often include complex decisions about resources, inputs from multiple users or a wide variety of requirements. To reduce complexity, intent-based solutions have been proposed for cloud management [6, 7].

In [6], an intent-based cloud service management framework is proposed, to automate or support the decision-making process for cloud resources. The framework takes an intent as input specified by a user in natural language. To support the user, hints are given about how to express the intent efficiently. For future developments, the authors describe a learning mechanism which learns from histories of user's requirements, such that more inputs are able to be parsed. The input in natural language is parsed into a JSON file containing the so called atom requirements. With this, the decision-making process is automated and parameters are calculated.

In [7], the authors propose a label management service for intent-based cloud management. An intent based interface must be simple and intuitive, but also portable by being decoupled from the specific details. To decouple intents from low-level details, the authors propose labels which are used in the process of specifying an intent. These labels are key-value pairs which capture the basic properties of the cloud entities, with the values being able to change dynamically. The intents are specified in natural language, in the form of subject-verb-object.

## 3.3 Blockchain Selection

A new field of application for intents is blockchain selection. The process of selecting the blockchain best suited for a specific use case can be demanding and requires expertise. In [8], the authors propose a CNL to support intent-based blockchain selection, which is applied in the Intent Refinement Toolkit (IRTK) [9, 28]. Section 2.4 covers the aspects of the IRTK in more detail. The CNL to input the intents supports different parameters, as shown in Table 2.1.

## 3.4 Discussion

Table 3.1 shows an overview of the works mentioned in this chapter and the method used for intent specification.

Table 3.1: Intent Specification Overview

| Reference | Area | Intent Specification |
|---|---|---|
| [26] | Network Management | Tuples |
| [27] | Network Management | CNL |
| [2] | Network Management | Natural Language |
| [6] | Cloud Management | Natural Language |
| [7] | Cloud Management | Natural Language |
| [9] | Blockchain Selection | CNL |

As can be seen in Table 3.1, most of the works presented in this section use either natural language or CNL for intent specification, in order to reach the level of abstraction desired with intents. [2] proposes to use a chat interface with machine learning to translate an intent into an intermediate intent representation, before refining it further into policies. In [6], the authors propose to use natural language with hints to support users expressing their requirements. Additionally, a learning mechanism is proposed. In contrast to natural language, a CNL-based intent can be refined much more easily, as a controlled vocabulary and grammar is used.

To conclude, the use of natural language for intent specification makes the intent refinement process more complex, having to deal with vagueness and ambiguities of natural language. However, with the employment of a CNL, the complexity is transferred to the user who has to understand and follow the rules in order to specify an intent. To support users, hints and suggestions could be given during the specification process.

# Chapter 4

# Blockchain Intent Management System

In this chapter, the Blockchain Intent Management System (`BIMS`) is described. The chapter details `BIMS`' design in Section 4.1, its implementation, including employed technologies, the extensions made in the IRTK [9] and the integration with the Policy-based Blockchain Selection framework *PleBeuS* [10] in Section 4.2.

## 4.1 `BIMS` Design

In this section the design and the architecture of the system is described. Figure 4.1 gives an overview over the system design with its components and actors. The primary actor is the user itself. A possible secondary actor could be a web service fetching and updating the currency exchange rates in the `BIMS`' database.

The remainder of this section explains the system components in more detail.

- **Graphical User Interface (GUI):** The GUI acts as the entry-point for a user, the main actor of the system. The GUI is decoupled from the rest of the system in order to have a modular design. The GUI is only responsible for displaying the outcome of actions from the backend in a user-friendly and intuitive way.
- **REST API:** As modularity is a key focus in the system design, the backend and the fronted (*i.e.,* the GUI) are decoupled and use a Representational State Transfer (RESTful) API as an interface to communicate. HTTP requests from the frontend are handled and passed on to other components using the JSON (JavaScript Object Notation) format to represent the data. For a detailed documentation of the endpoints of the API see Table 4.1.
- **User Manager:** The User Manager is a component of the backend which handles everything user-related. In more detail, the User Manager creates new users and stores them to the database. Moreover, it handles authentication with token

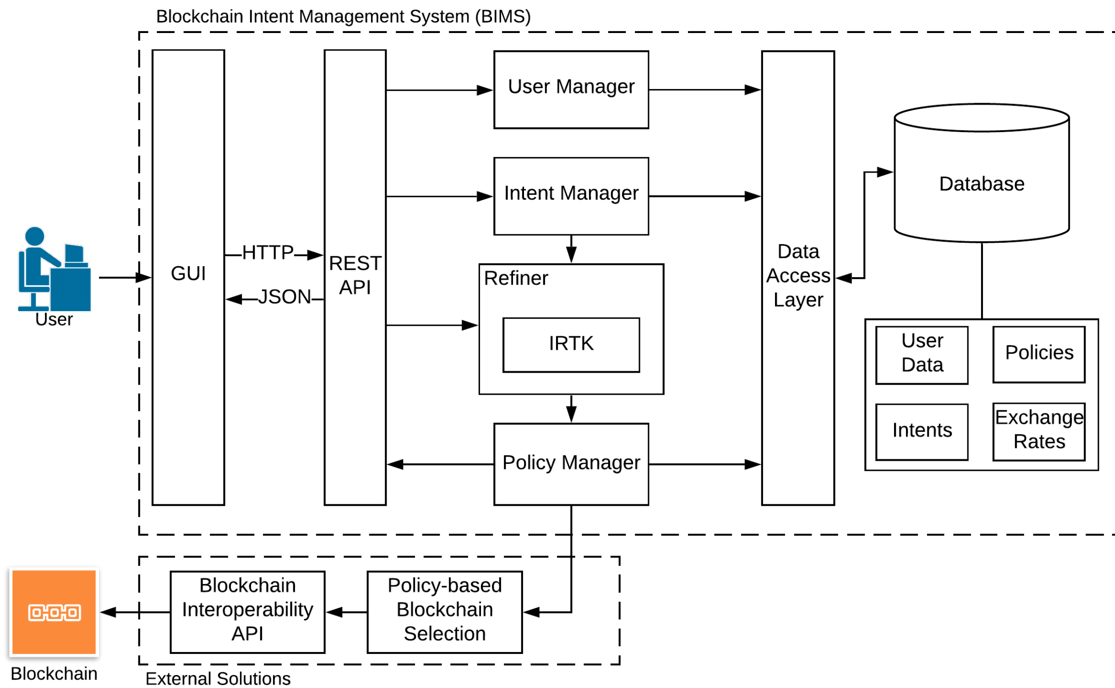


Figure 4.1: `BIMS` architecture

management, *e.g.*, if a user is logged in with correct credentials, it sends an automatically generated token to the frontend which is used to authenticate a user and manage access.

- **Intent Manager:** The Intent Manager component is responsible for the CRUD (Create, Read, Update, Delete) operations of intents. It handles requests from the frontend concerning intents and manages intents in the database. In case an intent is created or updated, the Intent Manager passes the intent to the Refiner component.

- **Refiner:** The Refiner component acts as a wrapper for the IRTK. It uses its functionalities to refine an intent and handle errors (in case an intent is not valid). Once refined and valid, the resulting policies are passed on to the Policy Manager. The Refiner also exposes an API endpoint for parsing an intent which refines an intent, but does not store or create anything. This is used in the frontend to check if a user input is a valid intent or give the user text suggestions.

- **Policy Manager:** The Policy Manager component handles the managing of policies received from the Refiner. It stores them to the database and exposes a read-only API endpoint. A policy cannot be created, changed or deleted directly via the API, only via an intent to ensure that the a policy always matches the refined intent. Therefore, the policy API endpoint is designed to be read-only. Furthermore, the policy manager integrates *PleBeuS*, a policy-based blockchain selection framework, by sending the policies to the external system .

- **Data Access Layer:** The Data Access Layer acts as an interface for the database,

leading to more modularity. The database is decoupled from the backend, allowing it to be changed independently.

- **Database:** The Database stores data from the system. For detailed information regarding the database entities and relationships, refer to Section 4.1.1.

### 4.1.1 Database

The database stores the data necessary in BIMS, which comprises Users, Intents, Policies and Currencies. The entity-relationship diagram of the BIMS' database is depicted in Figure 4.2 and its main entities described in the next items.

- **User:** A User type consists of a `username` which is a Primary Key (PK), and a `password`. The passwords are not stored in plain text for security reasons.
- **Intent:** An Intent has as primary key an ID. The User type is linked to an Intent as a Foreign Key (FK) in the Intent. Further, there are two timestamps for when an intent was created and updated. Finally, the intent itself is stored in a field called `intent_string`.
- **Policy:** A Policy also has an ID as primary key. It is linked to an Intent also by a foreign key and there are two timestamps like in the Intent. The field `pbs_id` stores the PleBeuS ID, which is needed when a policy is updated or deleted from PleBeuS. The other fields come from the IRTK. The `currency` field is a foreign key for a Currency type, although in the IRTK the currency of a policy is always USD. The `blockchain_pool` field consists of a set of blockchains, which is stored in form of a byte-string, allowing a Python set to be easily stored in the database.
- **Currency:** The Currency type has as primary key a three character currency code. Further, in the field `exchange_rate` the conversion rate to USD is stored. This is used by the IRTK to convert different currencies to USD.

#### Relationships

In Figure 4.2, the connections between the different entities denote the relationships they present. A User can have zero or many Intents, while an Intent can belong to one and only one User. If a User is deleted, so are all Intents belonging to the User.

An Intent can be refined into one or many Policies, but at least one. On the other hand, a Policy can belong to one and only one Intent. If an Intent is deleted, all Policies related to the Intent are deleted as well. A Policy has one and only one Currency, while a Currency can be in zero or many Policies.

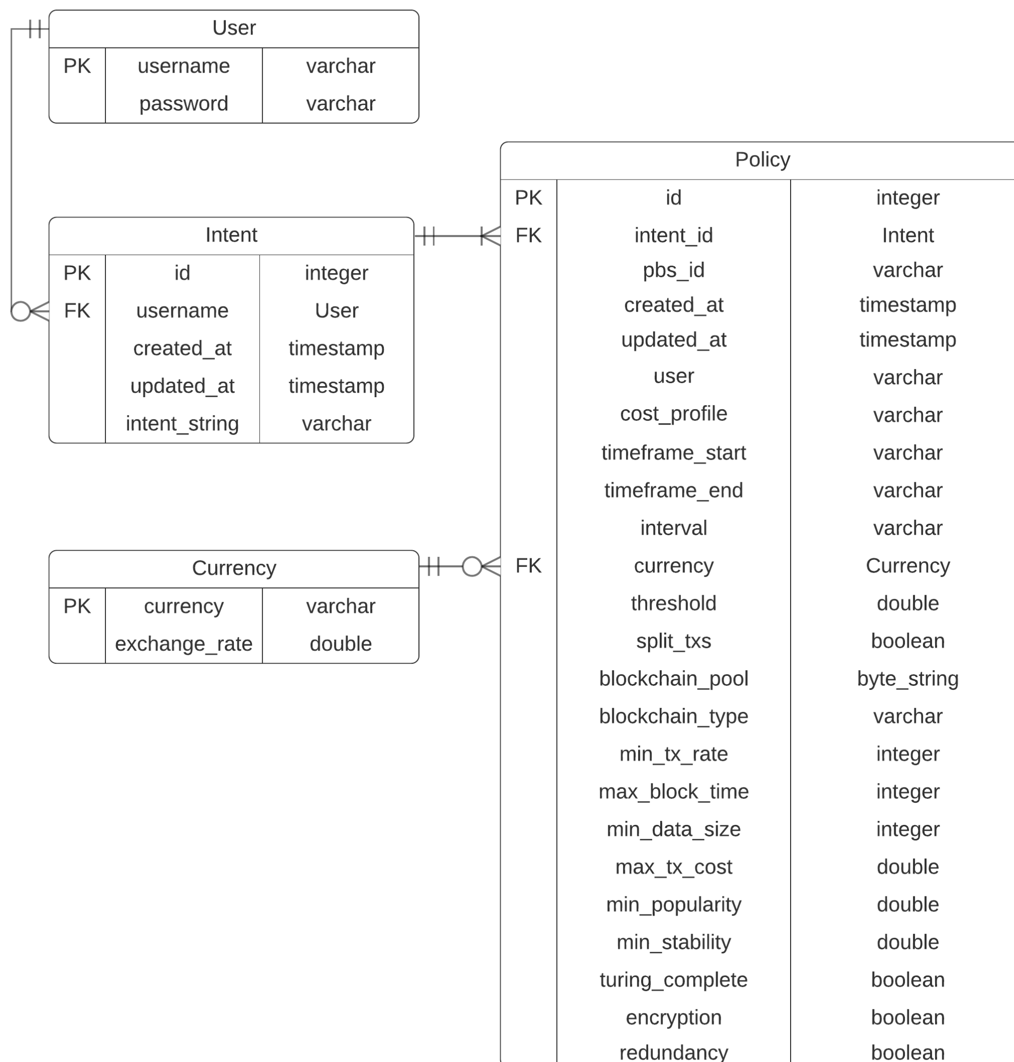


Figure 4.2: Entity-relation diagram

### 4.1.2 REST API

Table 4.1 documents the REST API exposed by `BIMS`. The first column with the title "Path" shows the different URLs used for the endpoints.

The column with the title "Header" shows which headers have to be sent in a request. The Authorization header consists of "`Token` " plus a token provided by the system. The value of the Content-Type header is `application/json`. The column with the title "Response Status" refers to HTTP status codes and their associated meanings.

### 4.1.3 User Interface

A strong focus during the design of the GUI was set on user experience. As the main goal of the system is to make a complicated process more accessible, a system which is

Table 4.1: REST API documentation

| Path | Headers | Method | Request Body | Description | Res. Status | Response Body |
|---|---|---|---|---|---|---|
| /api/auth/user | Authorization | GET | - | Returns the user with the provided token. | 200 | { id, username } |
| /api/auth/register | Content-Type | POST | { username, password } | Creates a user. | 201 | { user: { id, username }, token } |
| /api/auth/login | Content-Type | POST | { username, password } | Checks credentials and returns a token if valid. | 200 / 400 | { user: { id, username }, token } |
| /api/auth/logout | Authorization, Content-Type | POST | {} | Invalidates the token. | 204 / 401 | {} |
| /api/intents/ | Authorization, Content-Type | POST | { intent_string } | If valid, creates an intent. | 201 / 400 / 401 | { id, created_at, updated_at, intent_string, username } |
| /api/intents/<br>/api/intents/{id}/ | Authorization | GET | - | Returns a list of intents or a single intent belonging to the user. | 200 / 401 | { [ intents ] } |
| /api/intents/{id}/ | Authorization, Content-Type | PUT | { intent_string } | If valid, updates the intent. | 200 / 400 / 401 | { id, created_at, updated_at, intent_string, username } |
| /api/intents/{id}/ | Authorization | DELETE | - | Deletes the intent. | 204 / 401 | {} |
| /api/policies?intent_id={id} | Authorization | GET | - | Returns policies corresponding to the intent specified in the URL. | 200 / 401 | { [ policies ] } |
| /api/parser | Authorization, Content-Type | POST | { intent_string } | Parses an intent. If not valid, returns status 200, if valid 204. | 200 / 204 / 401 | { message, expected: [ ] } |

difficult to use would defeat its purpose. To this end, widely used design patterns were applied, such as a navigation bar at the top of the screen.

The GUI was designed using the principle of *wireframes* [29]. Thus, the page structure of the GUI was planned, as was the structure of the elements on each page. An example of a wireframe can be seen in Figure 4.3 for the Intent Details page.

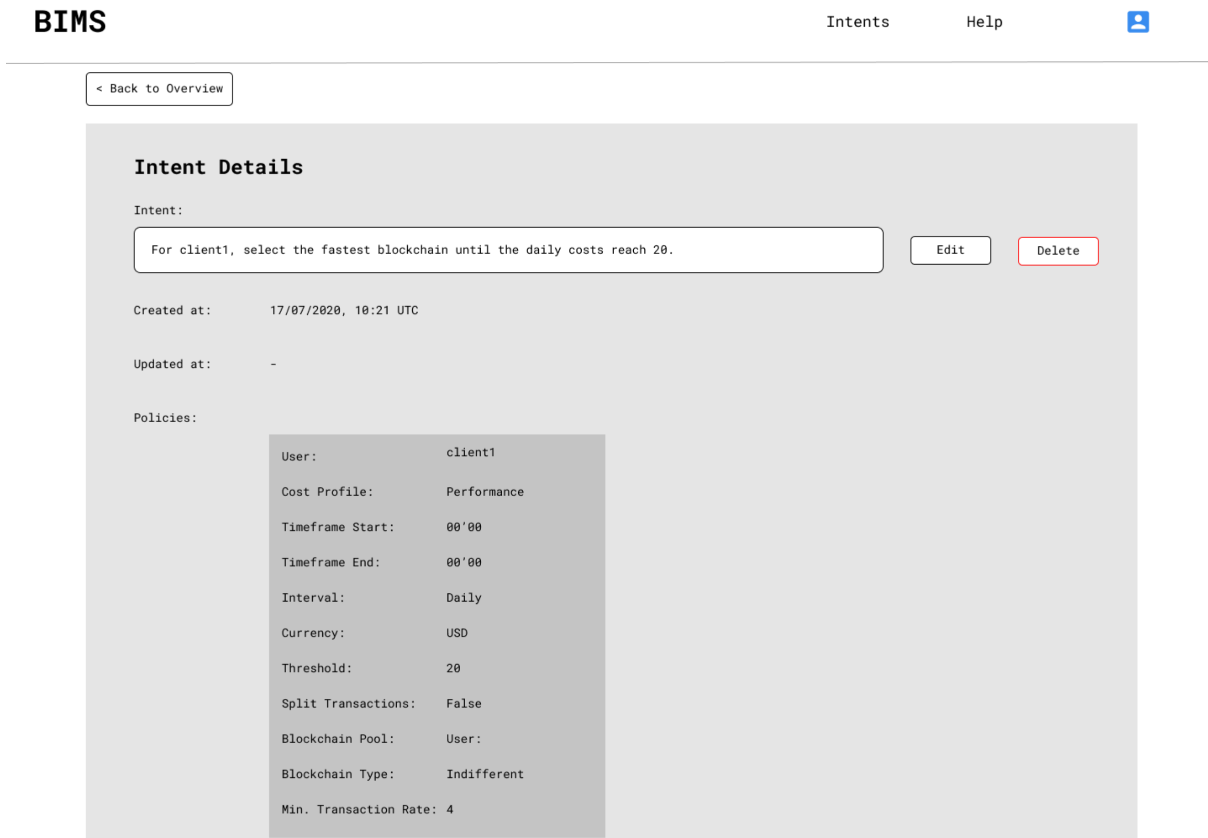


Figure 4.3: Wireframe for Intent Details page

To support users formulating a valid intent and to remove the need for extensive IRTK knowledge, it was decided that a text-field with text suggestions would be the best solution. For an input, the user is given a selection of valid next words. Thus, a user gets immediate feedback on his input and the text suggestions guide the user in the right direction, while still keeping the intuitiveness of natural language. If the user's input is not valid, no suggestions are shown, instead an error message is displayed. On the same page, an explanation is provided to support users on solving the issue.

## 4.2 Implementation

In this section, the details of the implementation, technologies used and changes made to the existing IRTK code are shown and discussed.

### 4.2.1 Technologies

#### Django

The backend containing the main logic of the system was written in Django. Django is a Python web framework which allows for fast development and comes with many features already implemented such as user authentication [30]. As the IRTK is written in Python, the decision was made to use Python as well for easy integration.

In Django, the core elements are models, views and templates. A model maps to a database table and defines its fields and attributes. An example for a model can be seen in Listing 4.1. A view takes requests and returns responses and renders a template. In a template, HTML is generated dynamically. This resembles a Model-View-Controller architecture, the view being the controller and the template being the view. However, in this case React is used for the frontend; thus, templates and views are not used. With models, Django abstracts the database offering a data access layer [30].

```python
from django.db import models
from user_manager.models import User

class Intent(models.Model):
    username = models.ForeignKey(User,
                                 related_name="intents",
                                 on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    intent_string = models.TextField()
```

Listing 4.1: Django model example

A Django project is divided into so-called applications, each one being a Python package with all necessary components, *e.g.,* models, views, templates, and URLs. This division allows for a good structure of the code. The User Manager, Intent Manager, Refiner and Policy Manager component (see Figure 4.1) are all implemented as an application. Additionally, each application / component defines its endpoints for the REST API and handles the database through the data access layer from Django.

#### React

The frontend was implemented using React, a JavaScript library for user interfaces. React uses either functional or class-based components to display what is seen in the GUI. A syntax extension to JavaScript called JSX is used to describe the elements of the GUI, similar to HTML. A class-based component requires a `render()` method, where the JSX elements are rendered. Next to `render()`, there are several other optional lifecycle methods such as `constructor()`, `componentDidMount()` or `componenDidUpdate()`. Lifecycle methods are executed in a particular order and at particular times.

As input, a component accepts `props`, which stands for properties. As shown in the example in Listing 4.2, the component `IntentInputField` is rendered in a parent component and takes as props various different values or functions [31].

```
<IntentInputField
    expected={this.props.expected}
    checkIfValid={this.checkIfValid}
    isValid={this.state.isValid}
    onSubmit={this.createIntent}
    buttonText="Submit"
    initialValue="For "
    parserMessage={this.props.parserMessage}
/>
```

Listing 4.2: Example for `props`

Each component can have a state where different fields can be set and the state of the component can be updated during the lifecycle [31].

#### Redux

Additionally to the component-wide state from React, an application-wide state was needed to manage the state of the whole system, not just one component. For this, Redux was used to have a single source of truth.

In Redux, the *store* is the object that holds the state which cannot be changed directly. In order to change the state, an *action* has to be dispatched to the *store*. In a *reducer*, it is defined how the state should change in response to the specific *action* dispatched. Once the *action* has been dispatched and the state in the *store* has been updated, the new state is sent to the frontend components [32].

### 4.2.2 Intent Auto-Completion

In order to support the user during the formulation of an intent, text suggestions with auto-completion were implemented. Based on the user's input, a list with valid words is suggested beneath the input field, as shown in Figure 4.5. Initially, the uppermost suggestion is selected. With the arrow keys, different suggestions can be selected and with either *Tab*, *Enter* or by clicking on a suggestion, the word is auto-completed. If the user made a mistake, instead of suggestions an error message is shown, depicted in Figure 4.6. If the input is a valid intent, the input field's colour is changed to green and the button to submit or update becomes clickable, as shown in Figure 4.7.

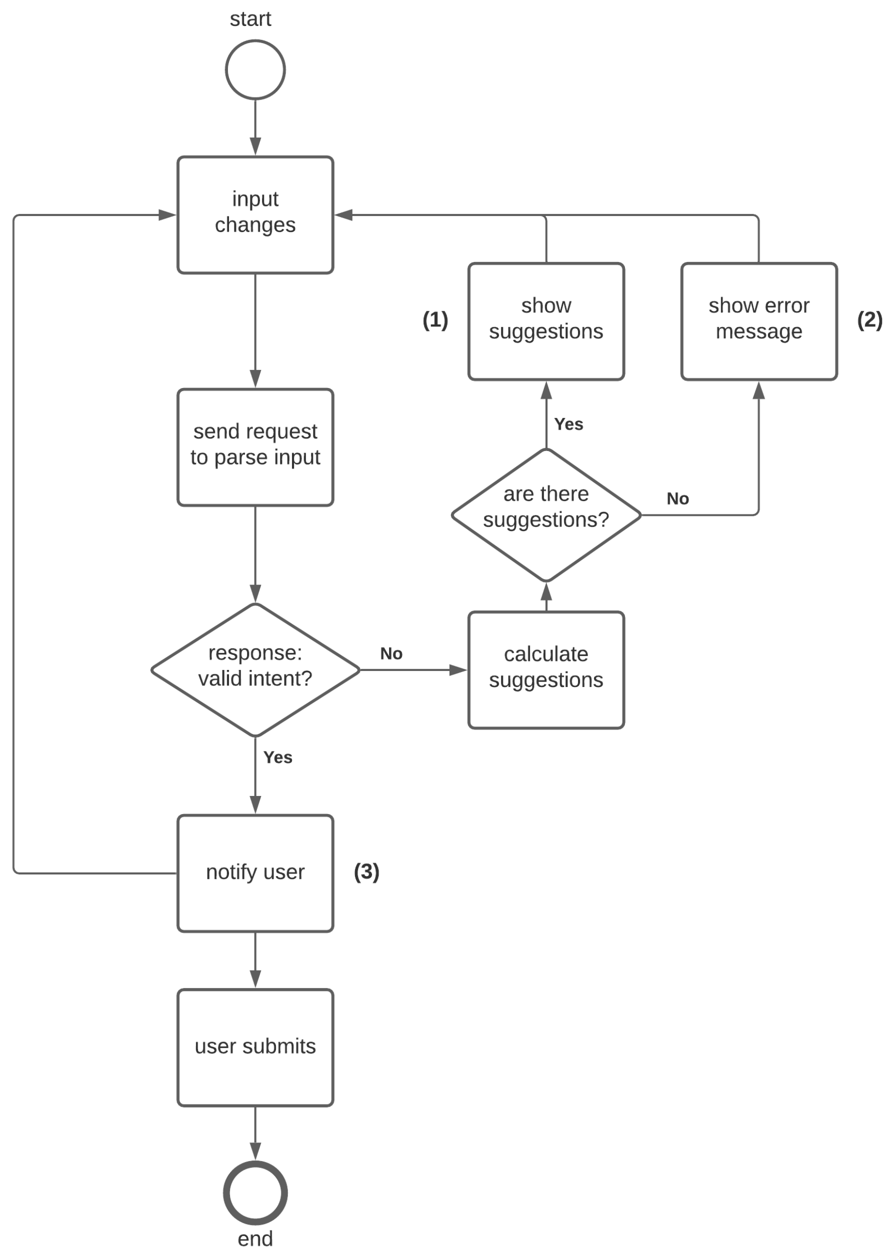


Figure 4.4: Intent parsing process

In Figure 4.4, the process of parsing the user input is displayed. Every time the input changes, a parsing request is sent to a dedicated API endpoint with the input. In the backend, this input is parsed by the IRTK. If the intent is not valid, the IRTK will raise an exception depending on the input. See Section 4.2.3 for the changes that had to be made to the IRTK for this mechanism to work. The backend sends a response with a message and a list of expected words to the frontend. There, the suggestions are filtered and displayed. If there are no suggestions, the message is displayed instead. If the input is a valid intent, the backend sends a response with the HTTP status code 204 (No Content). Numbers (1), (2) and (3) in Figure 4.4 correspond to Figures 4.5, 4.6 and 4.7.

For client1 select the fastest p

public
popular
private

Submit

Figure 4.5: **(1)** Text suggestions

For

Intent is incomplete

Submit

Figure 4.6: **(2)** Error message

For client1 select the fastest blockchain until the daily costs reach 10

Valid Intent

Submit

Figure 4.7: **(3)** Valid intent

```
class IntentParserAPI(generics.GenericAPIView):
    permission_classes = [
        permissions.IsAuthenticated
    ]
    serializer_class = ParserSerializer

    @staticmethod
    def post(request, *args, **kwargs):
        try:
            refine_intent(request.data.get('intent_string'))
        except (IllegalTransitionError,
                IncompleteIntentException,
                ValidationError) as error:
            return Response({
                'message': error.message,
                'expected': error.expected
            }, status=status.HTTP_200_OK)
        except Currency.DoesNotExist:
            return Response({
                'message': 'Currency was not found',
                'expected': []
            }, status=status.HTTP_404_NOT_FOUND)

        return Response(status=status.HTTP_204_NO_CONTENT)
```

Listing 4.3: Parser API implementation

Listing 4.3 shows the implementation of the parser API endpoint. In Listing 4.4 and 4.5, an example for a parsing request and a corresponding response is shown. The list of expected words is then filtered considering the user's input, *e.g.,* if the user types a character, only the suggestions starting with this character are shown, as can be seen in Figure 4.5.

```
{
    "intent_string": "For client1 select the fastest"
}
```

Listing 4.4: Parsing request data

```
{
    "message": "Intent is incomplete",
    "expected": [
        "popular",
        "blockchain",
        "cheap",
        "private",
        "public",
        "fast",
        "stable"
    ]
}
```

Listing 4.5: Parsing response data

### 4.2.3 IRTK Extension

Minimal changes to the IRTK's code had to be made. Since the IRTK uses its own database for currency conversion, it would not have been well-designed to have two separate databases. Thus, the IRTK was changed to access the same database as the rest of the system for currency conversions.

For easier implementation of text suggestions, the exceptions raised by the IRTK were changed as well. A new field in the exceptions was added containing an array of words that were expected by the IRTK. This array is used in the GUI to show text suggestions for a user. Also, a new exception was created when an intent is valid, but not complete. Originally, the parser did not raise an exception if a valid but incomplete intent is parsed. This is because of the option to parse an intent in multiple parts [9]. However, if the parser does not return an intent, the IRTK was changed to raise an exception, as shown in Listing 4.6.

```python
def refine(raw_intent: str) -> Optional[List[Policy]]:
    parser = Parser()
    intent = parser.parse(raw_intent)

    if not intent:
        LOGGER.warning("Refiner: could not refine incomplete
                       or invalid intent")
        raise IncompleteIntentException("Intent is incomplete",
                                        set(parser._state._transitions))

    policies = TRANSLATOR.translate(intent)
    return policies
```

Listing 4.6: `IncompleteIntentException` in `refine` function

### 4.2.4 PleBeuS Integration

The Policy-based Blockchain Selection framework *PleBeuS* [10] was integrated into `BIMS`. When an intent is created, it is refined into policies. These policies are passed on to *PleBeuS*. For a more detailed explanation of *PleBeuS* see Section 2.5. Listings 4.7 and 4.8 present the two methods connecting `BIMS` to *PleBeuS*.

*PleBeuS* exposes a REST API, which is used to create, update and delete policies. When a policy is created or updated, a POST request is sent to the API, when it is deleted, a DELETE request is sent. *PleBeuS*' policies are linked to a user. Each user has one and only one default policy which is used as a backup policy if none of the other policies match the criteria of the selection. This posed a challenge for the integration, as in `BIMS`, no restrictions on default policies are enforced.

If a non-default policy for a new user is created, a default policy has to be automatically created in *PleBeuS* prior to the non-default policy. As the automatically created default policy was not created with an intent in `BIMS`, it cannot be edited with an intent either. However, if for a new user the first intent which is created results in a default policy, this policy is linked to an intent. Thus, if this intent is updated, the default policy is updated as well. If a second default policy is created, an error message is shown. Further, if the intent resulting in the default policy is tried to be changed to a non-default intent, an error message is shown to the user.

In the backend configuration file `settings.py`, the option `USE_PLEBEUS` was added to configure whether to use *PleBeuS* or not. If the option to use *PleBeuS* is set to `False`, no requests will be sent. Further, the address for *PleBeuS* can be specified in `PLEBEUS_URL`.

The method `save_policy` is used for *(a)* creating and *(b)* updating a policy in *PleBeuS*. In case of *(a)*, a new policy is to be created, the input `pbs_id` is an empty string, while in the case of *(b)*, updating an existing policy, a correct `id` from *PleBeuS* has to be provided. First, a GET request is sent to see if the user already exists in *PleBeuS*. If it does, it can be assumed that a default policy for this user already exists. If the user does not exist and the policy is not a default policy, a POST request is sent to create a default policy before sending a second request with the new policy. Finally, the `id` of the new policy from *PleBeuS* is returned.

```
def save_policy(self, policy, pbs_id: str) -> str:
    """Makes a POST request to PleBeuS. Either creates a new Policy
    or updates an existing one if a pbs_id is provided.
    Returns the pbs_id"""

    if not settings.USE_PLEBEUS:
        return ''
    try:
        get_user_response = requests.get(self.pbs_url
                                         + '/policies/'
                                         + policy.user)
        if get_user_response.status_code == 404 and
            not self.__is_default_policy(policy):
            # need to create a default policy first
            default_p=self.__construct_default_policy_data(policy.user)
            default_p_response = requests.post(self.pbs_url +
                                               '/api/policies',
                                               default_p,
                                               headers=self.headers)
            if default_p_response.status_code != 201:
                # error when creating default policy
                raise PlebeusException(
                        json.loads(default_p_response.text).get(
                                                    'message'))

        data = self.__construct_policy_data(policy, pbs_id)
        response = requests.post(self.pbs_url + '/api/policies',
                                 data, headers=self.headers)
        if response.status_code != 201:
            raise PlebeusException(json.loads(response.text).get(
                                                    'message'))

        return json.loads(response.text).get('policy').get('_id')

    except ConnectionError:
        raise PlebeusException('Connection to PleBeuS failed')
```

Listing 4.7: PleBeuS integration: method to save policy

```
def delete_policy(self, pbs_id: str) -> None:
    """Makes a DELETE request to PleBeuS."""
    if not settings.USE_PLEBEUS:
        return
    try:
        requests.delete(self.pbs_url + '/api/policy/' + pbs_id)
    except ConnectionError:
        raise PlebeusException('Connection to PleBeuS failed')
```

Listing 4.8: PleBeuS integration: method to delete policy

The method `delete_policy` takes an `id` from *PleBeuS* as input and sends a DELETE request. Note that the response from *PleBeuS* is not checked if the deletion was successful.

This is because it should be possible to delete an intent even if the deletion in *PleBeuS* failed, for instance if the policy had already been manually deleted in *PleBeuS*.

It should be mentioned that such an integration is one-way only. This means that the changes made in `BIMS` will be reflected in *PleBeuS*. However, if any changes are made directly in *PleBeuS* (*e.g.,* updating policy parameters), these are not going to be reflected in the intent. This caveat is due to the fact that the IRTK refinement approach is top-down, refining a high-level policy (*i.e.,* an intent) to a low-level policy. Thus, the management should be made directly in `BIMS`, without direct interaction with *PleBeuS*.

# Chapter 5

# Evaluation and Discussion

In this chapter, the proposed system design and implementation are evaluated and discussed. The testing of the implementation of `BIMS` is described in Section 5.1. Following in Section 5.2, the usability of the system is evaluated by discussing case studies. Lastly, in Section 5.3 a discussion of the general benefits of the proposed solution concludes this chapter.

## 5.1 Software Testing

In order to find software defects, automated unit testing and integration testing was done during the development of `BIMS`. These tests were employed to continuously evaluate `BIMS` during its implementation, helping to identify code errors and wrong responses in a straightforward, quick and repeatable manner. The continuous implementation of test cases allowed for a fast evaluation whether the system behaves as expected for instance after a refactorization of one component.

All unit tests focus on one atomic part of the system, such that the correctness of every part can be evaluated. This included mocking external services such as *PleBeuS*, to get a prefabricated and controlled response. As every test case only tests one atomic part, if a test fails, the error can quickly be identified and fixed. For further evaluation of the system, integration testing was applied. This allowed for not just testing isolated units, but the system as a whole.

For example, the test case in Listing 5.1 evaluates a POST request to create an intent. The assertion in line 10 evaluates/asserts if the response has the expected HTTP status code. In line 12, the second assertion statement checks the number of intents created in the database. Finally, in line 14 the `username` field of the new intent is evaluated for correctness.

```
def test_post_intent(self):
    """test POST request of a valid intent"""
    self.client.credentials(HTTP_AUTHORIZATION='Token ' + self.token)
    intent = 'For client1 select the fastest Blockchain until the daily
              costs reach CHF 20'
    data = {'intent_string': intent}

    response = self.client.post(self.url, data, format='json')

    self.assertEqual(response.status_code, status.HTTP_201_CREATED,
                     'Incorrect status code')
    self.assertEqual(Intent.objects.count(), 1, 'Incorrect number of
                                                  intents')
    self.assertEqual(Intent.objects.get().username.id, self.user_id,
                     'Incorrect user_id in intent')
```

Listing 5.1: Example Unit Test

## 5.2 Case Studies

To evaluate the usability of the system, case studies in Table 5.1 will be discussed. The case studies are formulated in form of user stories and aim to cover the different use cases of BIMS from the user's perspective. In the following section, it is explained how the requirements described in the user stories are met.

Table 5.1: User stories

| Nr. | User Story |
|---|---|
| 1 | As a non-registered user, I want to be able to register with credentials in order to be able to log in. |
| 2 | As a non-logged in user, I want to be able to enter my credentials in order to log in. |
| 3 | As a user, I want to see all intents I created in an overview. |
| 4 | As a user, I want to be able to create a new intent. |
| 5 | As a user, I want to be able to see the policies which a specific intent is refined to, in order to see the effect of the intent. |
| 6 | As a user, I want to be able to delete an existing intent. |
| 7 | As a user, I want to be able to edit an existing intent. |
| 8 | As a user, I want to have clear intent parsing error messages, such that I understand why a specific process failed. |
| 9 | As a user, when I create an intent, I want to have text suggestions, such that it is easier to create a valid intent. |
| 10 | As a user, I want to have immediate feedback on my actions, such that I can act accordingly. |
| 11 | As a user, I want to have a page with information and guides in order that I can learn how to use and understand the system properly. |

**1: As a non-registered user, I want to be able to register with credentials in order to be able to log in.**

This requirement was accomplished by implementing authentication and registration. On a registration page (see Figure 5.1), users can register with a new account by entering a unique username and a password. The user is prompted to enter the password twice to ensure the password was correctly entered. In case a username is chosen which is already taken or the password fields do not match, an error message is displayed.

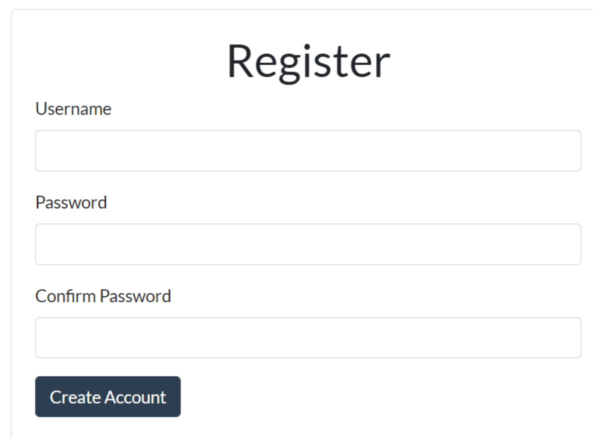


Figure 5.1: Registration form

**2: As a non-logged in user, I want to be able to enter my credentials in order to log in.**

As can be seen in Figure 5.2, on the login page, the user can enter a username and a password. In the backend, the credentials are validated and a token is returned if the credentials are valid. If the credentials are invalid, an error message is shown. The token is necessary for all requests for which you need to be authenticated. This ensures that for every action, a user has to be authenticated.

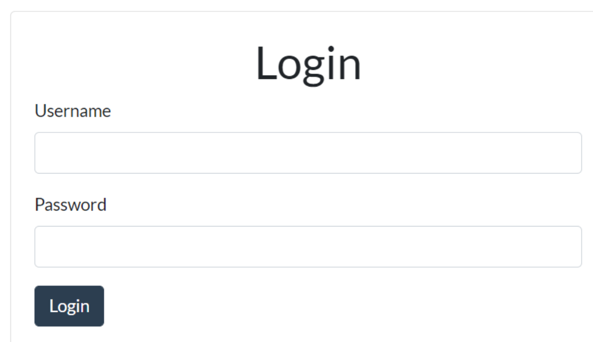


Figure 5.2: Login form

**3: As a user, I want to see all intents I created in an overview.**

As shown in Figure 5.3, the users see an overview listing all their intents with the time when they were created. Next to each intent, a button is displayed which redirects to the Intent Details page.

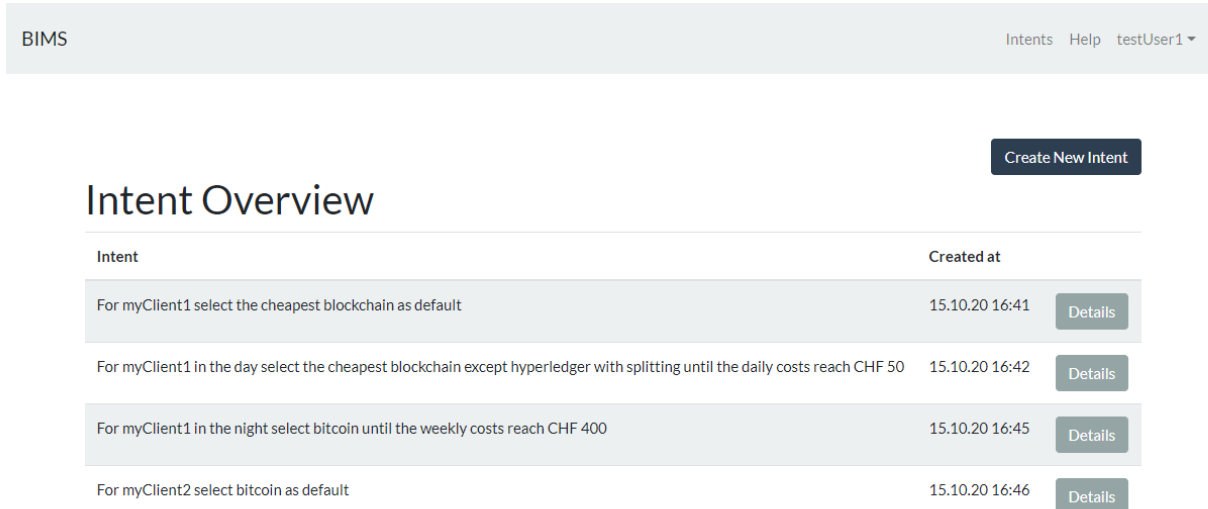


Figure 5.3: Intent Overview page

**4: As a user, I want to be able to create a new intent.**

To create a new intent, on the Intent Overview page a user can click the button in the top right corner. The user will be redirected to the Intent Creation page seen in Figure 5.4. The page contains an input field where the intent is formulated and an explanation on intent creation below. The button to submit the intent can only be clicked if the intent is valid.

**5: As a user, I want to be able to see the policies which a specific intent is refined to, in order to see the effect of the intent.**

In order to see details about an intent, users have the possibility to access the Details page of an intent as shown in Figure 5.5. In addition to the information shown on the Overview page, a timestamp of the last time the intent was updated is shown as well as the policies refined from the intent. In case of a multi-user intent, the policies for all users are shown here. With this information, the user sees what effect the intent has on the policy and finally on the selection of a blockchain.

Figure 5.4: Intent Creation page

**6: As a user, I want to be able to delete an existing intent.**

On the Intent Details page shown in Figure 5.5, there is a button to delete an intent. To prevent unintentional deletion, a pop-up window is shown where a user is prompted to confirm or cancel the deletion of the intent.

**7: As a user, I want to be able to edit an existing intent.**

Also on the Intent Details page, there is a button to edit an intent. The user will be redirected to the Intent Edit page. The Intent Edit page is structured very similarly to the Intent Creation page shown in Figure 5.4.

**8: As a user, I want to have clear intent parsing error messages, such that I understand why a specific process failed.**

During the intent specification process, an intent is parsed by the IRTK. If there is an error in the parsed intent, the respective error message is shown to the user. Related to this, the user is given text suggestions with the aim to help the user. If there is a different error during the process, an alert is shown to the user as shown in Figure 5.6.

BIMS Intents Help testUser1

Back

Intent Details

Edit Delete

| | |
|---|---|
| Intent: | For myClient2 and myClient3 select bitcoin as default |
| Created at: | 15.10.20 16:46 |
| Updated at: | 15.10.20 17:00 |
| Policies: | |

| Policy 1 | | Policy 2 | |
|---|---|---|---|
| User: | myclient2 | User: | myclient3 |
| Cost Profile: | CostProfile.ECONOMIC | Cost Profile: | CostProfile.ECONOMIC |
| Timeframe Start: | Time.DEFAULT | Timeframe Start: | Time.DEFAULT |
| Timeframe End: | Time.DEFAULT | Timeframe End: | Time.DEFAULT |
| Interval: | Interval.DEFAULT | Interval: | Interval.DEFAULT |
| Currency: | USD | Currency: | USD |
| Threshold: | 0 | Threshold: | 0 |
| Split Transactions: | False | Split Transactions: | False |
| Blockchain Pool: | • BITCOIN | Blockchain Pool: | • BITCOIN |
| Blockchain Type: | BlockchainType.INDIFFERENT | Blockchain Type: | BlockchainType.INDIFFERENT |
| Min. Trans. Rate: | 4 | Min. Trans. Rate: | 4 |
| Max. Block Time: | 600 | Max. Block Time: | 600 |
| Min. Data Size: | 20 | Min. Data Size: | 20 |
| Max. Trans. Cost: | 0 | Max. Trans. Cost: | 0 |
| Min. Popularity: | 0 | Min. Popularity: | 0 |
| Min. Stability: | 0 | Min. Stability: | 0 |
| Turing Complete: | False | Turing Complete: | False |
| Encryption: | False | Encryption: | False |
| Redundancy: | False | Redundancy: | False |

Figure 5.5: Intent Details page

**9: As a user, when I create an intent, I want to have text suggestions, such that it is easier to create a valid intent.**

As explained in Section 4.2.2, text suggestions during the specification of an intent were implemented. If a user follows the suggestions given, a valid intent will be specified. Thus, if no suggestions and an error message is given instead, the user knows that there is a mistake.

**10: As a user, I want to have immediate feedback on my actions, such that I can act accordingly.**

This requirement is met in different ways. During the specification of the intent, users get immediate feedback with every character typed, as the input is parsed every time it changes. Also, messages and alerts are shown after actions, such as successful logout or

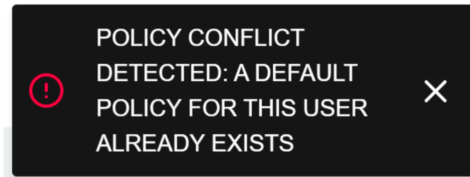


Figure 5.6: Alert example

deletion of an intent. This feedback keeps users informed about the state of the system, *i.e.*, if the outcome of an action was successful or if there has been an error.

**11: As a user, I want to have a page with information and guides in order that I can learn how to use and understand the system properly.**

For users to get information about the system, a Help page was implemented as shown in Figure 5.7. On this page, explanations on how to use the system are given, as well as general information about the purpose and background of `BIMS`.

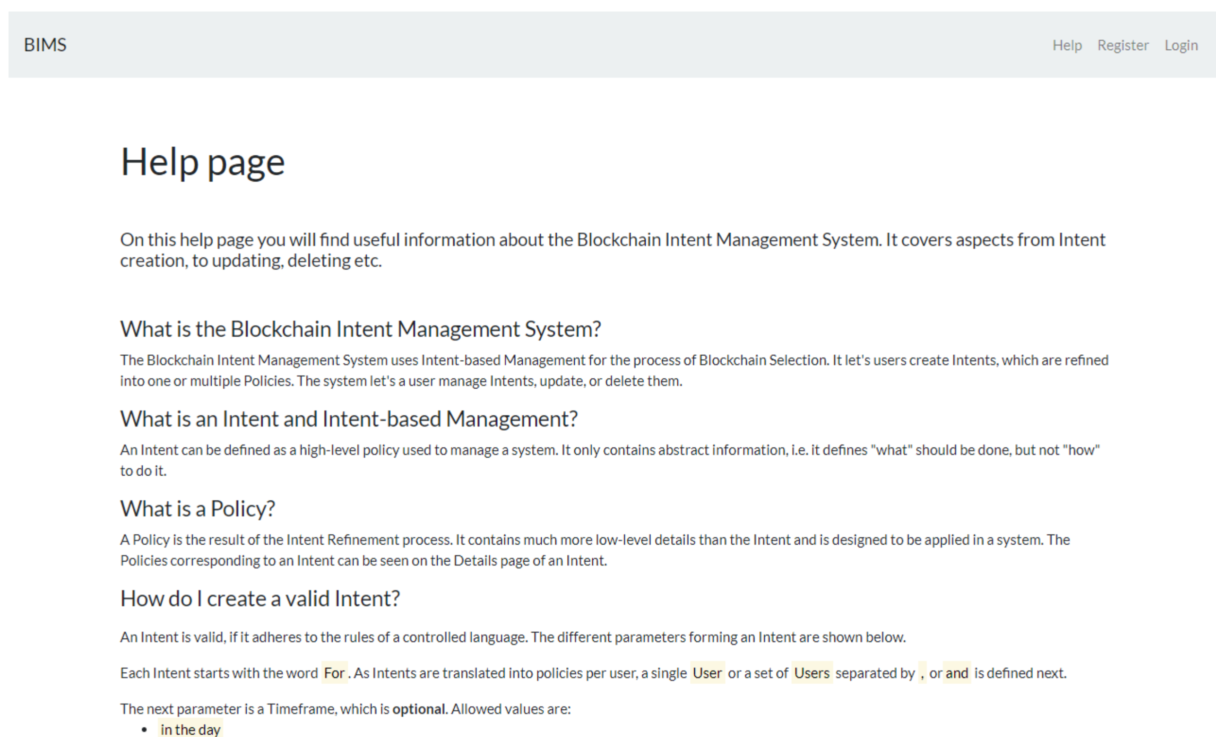


Figure 5.7: Help page

## 5.3 Discussion

As presented in Section 5.2, the requirements listed during the design of the system were successfully met. Thus, the question arises as to what benefits there are to end-users of a system designed as described in this thesis.

There are different expected benefits for users of `BIMS`. For example, by using intents, low-level details are abstracted from the user. This leads to less technical understanding required from the user, allowing access to more users with a less technical background. As observed in a survey with users in [9], a representation of a policy in natural language is perceived as more intuitive compared to a representation in pseudo-code. Thus, the usage of intents makes the process of blockchain selection more intuitive.

Furthermore, users benefit from support with text suggestions and auto-completion. The text suggestions lead users to a valid specification of an intent. Moreover, the suggestions and auto-completion are implemented and designed similarly to other applications, resulting in an intuitive usage. Moreover, users have immediate feedback from the system. As a consequence, the users know the state of the application, if an action failed or succeeded.

To summarize, `BIMS` provides two main benefits: *(i)* abstraction of details and *(ii)* intuitive usage to support users. Both of these benefits lead to a simplification of the process of blockchain selection and a less error-prone usage.

# Chapter 6

# Summary and Future Work

This thesis presented the design, implementation, and evaluation of a Blockchain Intent Management System called `BIMS`. The proposed system simplifies the process of blockchain selection by using intent-based management. Users are able to specify an intent in a controlled natural language, which is refined by the Intent Refinement Toolkit (IRTK) into a policy, specifying parameters for the selection of a blockchain.

The proposed system exposes a Graphical User Interface (GUI), where users are able to manage intents, *i.e.,* create, read, update and delete them. During the specification of an intent, users are supported by text suggestions in order to facilitate the creation of a valid intent. Further, the user is able to see detailed information about an intent, *e.g.,* the policies which were refined from a specific intent. The GUI communicates via a REST API with the backend, where different components handle requests and act accordingly. `BIMS` integrates a Policy-based Blockchain Selection framework called *PleBeuS*, which takes the policies and applies selection algorithms to select a blockchain for incoming transactions. Currently, the integration is only implemented in one direction, *i.e.,* changes made in `BIMS` are reflected in *PleBeuS*, but not vice versa.

Further, based on the performed evaluation, it can be seen that `BIMS` addressed the user stories listed and that `BIMS` is able to simplify the specification of blockchain intents by providing an intent editor with text auto-completion and immediate feedback to the user.

## 6.1 Future Work

As `BIMS` uses the IRTK for intent refinement, the intent parameters are limited to those supported by the IRTK. The intent parameters of the IRTK could be extended to support more parameters such as more blockchains or more currencies. This would offer more flexibility for end-users with a manageable effort.

In the prototype described in this thesis, `BIMS` and *PleBeuS* are implemented as two different systems communicating via a REST API. This results in a number of issues, *e.g.,* reduced performance due to the API calls to an external service. Furthermore, the

integration of *PleBeuS* only works in one direction, changes made directly in *PleBeuS* are not reflected in `BIMS`. While `BIMS` implements authentication with secured user accounts, *PleBeuS* does not. Thus, a policy in *PleBeuS* can easily be edited or deleted by another user. Further, *PleBeuS* does not support all parameters of the IRTK. Namely, the parameters `encryption`, `redundancy`, `stable`, `popular` and `cheap` are ignored in *PleBeuS*. To address these issues, a complete integration of *PleBeuS* as an internal component of `BIMS` would improve the performance, integrity, correctness and usability of the system.

The GUI of `BIMS` could be extended as well, for instance with options to filter or sort the list of intents in the overview. If *PleBeuS* were integrated as an internal component, more information about policies could be displayed as well, for instance whether a policy is active or not. Furthermore, `BIMS` could be extended to periodically fetch current exchange rates from an external service. However, this would lead to more external dependencies. Also, existing policies would have to be updated with the current exchange rates.

# Bibliography


[1] M. H. Behringer, M. Pritikin, S. Bjarnason, A. Clemm, B. E. Carpenter, S. Jiang, and L. Ciavaglia, "Autonomic Networking: Definitions and Design Goals." RFC 7575, June 2015.

[2] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville, "Refining Network Intents for Self-Driving Networks," in *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, (New York, NY, USA), pp. 15–21, Association for Computing Machinery, 2018.

[3] A. Clemm, L. Ciavaglia, L. Granville, and J. Tantsura, "Intent-Based Networking - Concepts and Overview," 2019. Work in Progress, `https://tools.ietf.org/id/draft-clemm-nmrg-dist-intent-03.html`, Last visit June 26, 2020.

[4] Q. Sun, W. S. LIU, and K. Xie, "An Intent-driven Management Framework," Internet-Draft draft-sun-nmrg-intent-framework-00, Internet Engineering Task Force, July 2019. Work in Progress.

[5] C. Li, O. Havel, W. S. LIU, A. Olariu, P. Martinez-Julia, J. C. Nobre, and D. Lopez, "Intent Classification," Internet-Draft draft-li-nmrg-intent-classification-03, Internet Engineering Task Force, Apr. 2020. Work in Progress.

[6] W. Chao and S. Horiuchi, "Intent-based Cloud Service Management," in *21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN 2018)*, (Paris, France), pp. 1–5, 2018.

[7] J. Kang, J. Lee, V. Nagendra, and S. Banerjee, "LMS: Label Management Service for Intent-driven Cloud Management," in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2017)*, (Lisbon, Portugal), pp. 177–185, May 2017.

[8] E. J. Scheid, P. Widmer, B. Rodrigues, M. Franco, and B. Stiller, "A Controlled Natural Language to Support Intent-based Blockchain Selection," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC 2020)*, (Toronto, Canada), pp. 1–9, IEEE, May 2020.

[9] P. Widmer, "Design and Implementation of an Intent-based Blockchain Selection Framework," Master's thesis, University of Zurich, Jan. 2020.

[10] E. J. Scheid, D. Lacik, B. Rodrigues, and B. Stiller, "PleBeuS: a Policy-based Blockchain Selection Framework," in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2020)*, (Budapest, Hungary), pp. 1–9., Apr. 2020.

[11] R. Boutaba and I. Aib, "Policy-based Management: A Historical Perspective," *Journal of Network and Systems Management*, vol. 15, no. 4, pp. 447–480, 2007.

[12] D. C. Verma, "Simplifying Network Administration Using Policy-based Management," *IEEE Network*, vol. 16, no. 2, pp. 20–26, 2002.

[13] S. Waldbusser, J. Perry, S. Herzog, M. A. Carlson, D. J. M. Schnizlein, B. Quinn, M. Scherling, A. Westerinen, A. Huynh, and J. Strassner, "Terminology for Policy-Based Management." RFC 3198, Nov. 2001.

[14] S. Davy, B. Jennings, and J. Strassner, "The Policy Continuum - A Formal Model," *in Proc. of the 2nd IEEE International Workshop on Modelling Autonomic Communications Environments, MACE*, pp. 65–79, Jan. 2007.

[15] D. R. C. Moore, J. Strassner, E. J. Ellesson, and A. Westerinen, "Policy Core Information Model – Version 1 Specification." RFC 3060, Feb. 2001.

[16] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[17] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2009. `https://bitcoin.org/bitcoin.pdf`, Last visit June 26, 2020.

[18] D. Puthal, N. Malik, S. P. Mohanty, E. Kougianos, and G. Das, "Everything You Wanted to Know About the Blockchain: Its Promise, Components, Processes, and Problems," *IEEE Consumer Electronics Magazine*, vol. 7, no. 4, pp. 6–14, 2018.

[19] K. Wüst and A. Gervais, "Do you Need a Blockchain?," in *Crypto Valley Conference on Blockchain Technology (CVCBT 2018)*, (Zug, Switzerland), pp. 45–54, November 2018.

[20] S. King and S. Nadal, "PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake," Aug. 2012. `https://www.peercoin.net/whitepapers/peercoin-paper.pdf`, Last visit July 3, 2020.

[21] Ethereum Foundation, "Proof of Stake FAQs," 2020. `https://eth.wiki/en/concepts/proof-of-stake-faqs`, Last visit July 3, 2020.

[22] EOSIO, "EOSIO Consensus Protocol," 2020. `https://developers.eos.io/welcome/latest/protocol/consensus_protocol`, Last visit July 7, 2020.

[23] POA Network, "Proof of Authority: consensus model with Identity at Stake.," 2017. `https://medium.com/poa-network/proof-of-authority-consensus-model-with-identity-at-stake-d5bd15463256`, Last visit July 7, 2020.

[24] Sawtooth, "PoET 1.0 Specification," 2018. `https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html`, Last visit July 7, 2020.

[25] I. Trajkovska, "A glimpse of the Intent-based datacenter," May 2018. `https://gblogs.cisco.com/ch-tech/a-glimpse-of-the-intent-based-datacenter/`, Last visit July 10, 2020.

[26] Y. Elkhatib, G. Coulson, and G. Tyson, "Charting an intent driven network," in *13th International Conference on Network and Service Management (CNSM 2017)*, (Tokyo, Japan), pp. 1–5, Nov. 2017.

[27] E. J. Scheid, C. C. Machado, M. F. Franco, R. L. dos Santos, R. P. Pfitscher, A. E. Schaeffer-Filho, and L. Z. Granville, "INSpIRE: Integrated NFV-based Intent Refinement Environment," in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2017)*, (Lisbon, Portugal), pp. 186–194, May 2017.

[28] P. Widmer, E. J. Scheid, and B. Rodrigues, "Intent Refinement Toolkit (IRTK)," 2019. `https://gitlab.ifi.uzh.ch/scheid/irtk-code`, Last visit May 05, 2020.

[29] G. Chen, "What Is a Wireframe?," 2020. `https://productmanagerhq.com/what-is-a-wireframe/`, Last visit October 2, 2020.

[30] Django Software Foundation, "Django documentation," 2020. `https://docs.djangoproject.com/en/3.1/`, Last visit September 29, 2020.

[31] Facebook Inc., "React," 2020. `https://reactjs.org/`, Last visit September 29, 2020.

[32] Dan Abramov, "Redux," 2020. `https://react-redux.js.org/`, Last visit September 29, 2020.

# Abbreviations

| | |
|---|---|
| AI | Artificial Intelligence |
| API | Application Programming Interface |
| BIMS | Blockchain Intent Management System |
| CLI | Command Line Interface |
| CNL | Controlled Natural Language |
| CRUD | Create, Read, Update, Delete |
| DFA | Deterministic Finite Automaton |
| DMTF | Distributed Management Task Force |
| dPoS | Delegated Proof-of-Stake |
| ECA | Event-Condition-Action |
| FK | Foreign Key |
| GUI | Graphical User Interface |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| IBN | Intent-Based Networking |
| IBS | Intent-Based System |
| IETF | Internet Engineering Task Force |
| IRTK | Intent Refinement Toolkit |
| JSON | JavaScript Object Notation |
| JSX | JavaScript XML (React syntax extension to JavaScript) |
| MIB | Management Information Base |
| PBM | Policy-Based Management |
| PDP | Policy Decision Point |
| PEP | Policy Enforcement Point |
| PIB | Policy Information Base |
| PK | Primary Key |
| PMT | Policy Management Tool |
| PoA | Proof-of-Authority |
| PoET | Proof-of-Elapsed-Time |
| PoS | Proof-of-Stake |
| PoW | Proof-of-Work |
| P2P | Peer-to-Peer |
| REST | Representational State Transfer |
| SLA | Service Level Agreement |
| SSoT | Single Source of Truth |
| TTP | Trusted Third Party |

| | |
|---|---|
| URL | Uniform Resource Locator |
| VNF | Virtual Network Function |

# List of Figures


2.1 Policy continuum . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.2 Policy-based management architecture . . . . . . . . . . . . . . . . . . . 5

2.3 Intent lifecycle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

4.1 `BIMS` architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

4.2 Entity-relation diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4.3 Wireframe for Intent Details page . . . . . . . . . . . . . . . . . . . . . 22

4.4 Intent parsing process . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4.5 **(1)** Text suggestions . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

4.6 **(2)** Error message . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

4.7 **(3)** Valid intent . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

5.1 Registration form . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

5.2 Login form . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

5.3 Intent Overview page . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

5.4 Intent Creation page . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

5.5 Intent Details page . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

5.6 Alert example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

5.7 Help page . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

# List of Tables

2.1 Intent parameters of the IRTK . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.1 Intent Specification Overview . . . . . . . . . . . . . . . . . . . . . . . . . 15

4.1 REST API documentation . . . . . . . . . . . . . . . . . . . . . . . . . . 21

5.1 User stories . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

# Appendix A

# Installation Guidelines

## A.1 Code Structure

The following code structure is intended to find important files more easily. Thus, only important files for the setup are mentioned.

```
bims
    |___bims
    |   |   settings.py
    |
    |___frontend
    |
    |___intent_manager
    |
    |___policy_manager
    |
    |___refiner
    |   |___irtk
    |   |   |   config.py
    |
    |___user_manager
    |
    |   manage.py

README.md
requirements.txt
currencies.json
logging.conf
package.json
```

## A.2 Setup

Note that for different operating systems the commands are slightly different.

#### Requirements

- Python 3.8 or later
- PostgreSQL
- Node.js and npm

#### Dependencies

1. Create and activate a virtual environment

```
$ python -m venv venv
```

activate on UNIX:

```
$ source venv/bin/activate
```

activate on Windows

```
C:\> venv\Scripts\activate.bat
```

2. Install the dependencies:

```
(venv) $ pip install -r requirements.txt
```

#### Database

1. Create a postgres database

On Windows: start Postgres server (path to PostgreSQL files might be different)

```
C:\> pg_ctl -D "C:\Program Files\PostgreSQL\12\data" start
```

Start PostgreSQL interactive terminal:

```
$ psql -U postgres
```

Create a User:

```
# CREATE USER *username* WITH PASSWORD '*password*';
```

Create a database:

```
# CREATE DATABASE *database name* OWNER *username*;
```

Use `\q` to quit the PostgreSQL terminal

2. Update database settings in `settings.py`

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': '*database name*',
        'USER': '*username*',
        'PASSWORD': '*password*',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Note that commands using `manage.py` should be run from the base directory, *i.e.* use `bims/manage.py`.

Make migrations:

```
(venv) $ python bims/manage.py makemigrations
```

Migrate:

```
(venv) $ python bims/manage.py migrate
```

Load currencies in database:

```
(venv) $ python bims/manage.py loaddata currencies.json
```

### GUI

Install dependencies:

```
$ npm install
```

Build project:

```
$ npm run build
```

### Configure PleBeuS

In `bims/bims/settings.py`:

Use PleBeuS? `USE_PLEBEUS=True` / `USE_PLEBEUS=False`

Configure PleBeuS server address and port: `PLEBEUS_URL=''`

**Run Server**

1. Make sure database server is running

2. Run Django server:

```
(venv) $ python bims/manage.py runserver
```

By default, the server will run on 127.0.0.1:8000, however a different address and port can be specified.

**Run tests**

```
(venv) $ python bims/manage.py test user_manager.tests
(venv) $ python bims/manage.py test intent_manager.tests
(venv) $ python bims/manage.py test policy_manager.tests.tests
(venv) $ python bims/manage.py test policy_manager.tests.plebeus_tests
(venv) $ python bims/manage.py test refiner.tests.tests
(venv) $ python bims/manage.py test refiner.tests.parser_tests
(venv) $ python bims/manage.py test bims.integration_tests
```

Note that in order to run tests, the database user needs permission to create a test database in PostgreSQL.

## A.3 Troubleshooting

**OperationalError: could not connect to server: Connection refused**

```
django.db.utils.OperationalError: could not connect to server: Connection refused
could not connect to server: Connection refused
    Is the server running on host "localhost" (127.0.0.1) and accepting
    TCP/IP connections on port 5432?
```

It seems like the database cannot be accessed. Make sure the Postgres server is running at the specified address and port.

**KeyError: 'formatters'**

This is an issue with the IRTK logger. Run your command from the same directory as the `logging.conf` file.

**LookupError: Resource punkt not found**

```
LookupError: Resource punkt not found.
Please use the NLTK Downloader to obtain the resource:
>>> import nltk
>>> nltk.download('punkt')
Attempted to load tokenizers/punkt/PY3/english.pickle
```

This error message indicates that the punkt resource from the nltk data is missing. It is required for tokenizing the intents. To resolve the issue run:

```
(venv) $ python -m nltk.downloader punkt
```

# Appendix B

# Contents of the CD

- `BA-Sandro-Padovan.pdf` : PDF with the complete documented thesis.
- `thesis.zip` : LaTeX source files
- `BIMS.zip` : Code source files
- `Zusfsg.txt` : Abstract in German
- `Abstract.txt` : Abstract in English
- `MidtermPresentation.pptx` : Midterm presentation slides