Bror Tobias Jacobsen Hagehei

DefiFunds - A Dapp on Radix

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2023

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Bror Tobias Jacobsen Hagehei

DefiFunds - A Dapp on Radix

DefiFunds

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Abstract

In this master's thesis, a decentralized application (Dapp) called DefiFunds has been further developed. A proof of concept for this Dapp was created on Radix during my specialization project in the fall of 2022. The Radix network is still in its early days, with smart contracts estimated to go live on mainnet by the 31st of July, 2023. The final goal is to bring the Dapp to production. During this master's thesis, a hackathon was participated in, resulting in an honorable mention prize worth $3,500.

DefiFunds is a Dapp that connects fund managers with investors. The unique feature of it being decentralized is that no one, except the user, has access to withdraw their share of the fund. Neither DefiFunds nor the fund manager is in control of the funds, as they are securely stored in a smart contract. The fund manager is able to trade with the funds, but never withdraw them to himself.

During the course of the master's thesis, additional functionalities related to the smart contract have been developed. A functional, though not user-friendly, frontend has also been implemented. In addition to these technical improvements, a freelancer has contributed by crafting a design for DefiFunds.

This master's thesis begins with an introduction to distributed ledger technologies, and then focuses on Radix specifically. Next, it presents the design and implementation of the Dapp, followed by a section detailing the triumph achieved in the hackathon. Challenges associated with the development process are then discussed. Finally, the report concludes with two lists explaining what has been done, and what needs to be done in order for the Dapp to be complete.

Sammendrag

I denne masteroppgaven har en desentralisert applikasjon (Dapp) kalt DefiFunds blitt videreutviklet. Et konseptbevis for denne Dappen ble laget på Radix under mitt spesialiseringsprosjekt høsten 2022. Radix er fremdeles i sine tidlige dager, og smartkontrakter er anslått å komme til hovednettet den 31. juli 2023. Det endelige målet er å ferdigutvikle Dappen. Under denne masteroppgaven ble det deltatt i et hackathon, noe som resulterte i en hederlig omtale og en premie med omtrentlig verdi på 35 000 kr.

DefiFunds er en Dapp som kobler fondsforvaltere med investorer. Det unike med at Dappen er desentralisert er at ingen, bortsett fra brukeren, har tilgang til å ta ut sin andel av fondet. Hverken DefiFunds eller fondsforvalteren har kontroll over midlene, da de er trygt lagret i en smartkontrakt. Fondsforvalteren kan handle med midlene, men aldri ta dem ut til seg selv.

I løpet av masteroppgaven er det blitt utviklet flere funksjoner på smartkontrakten. En funksjonell, men ikke brukervennlig, frontend har også blitt implementert. I tillegg til disse tekniske forbedringene har en frilanser bidratt med å lage et brukergrensesnitt til DefiFunds.

Denne masteroppgaven begynner med en introduksjon til distribuerte ledger teknologier, og fokuserer deretter spesifikt på Radix. Den presenterer designet og implementeringen av Dappen, etterfulgt av en seksjon om hackatonet. Deretter diskuteres utfordringer knyttet til utviklingsprosessen. Til slutt konkluderer rapporten med to lister som forklarer hva som har blitt gjort, og hva som trenger å bli gjort for å ferdigstille Dappen.

Preface

This master's thesis is the final part of my master's degree in real-time systems. The thesis work was conducted at the Norwegian University of Science Technology and is a continuation of the work done in my specialization project. The master's thesis isn't directly relevant to my degree, but I chose the topic because of my strong interest in decentralized finance since 2018 and Radix since 2021. I was also eager to work on something that has a real chance of making an impact.

I would like to thank my supervisor Sverre Hendseth for his time. His guidance was not how I thought it would be in the beginning, but it ended up fitting me very well. His advice was rarely concrete and often explained using metaphors. I learned to deal with the lots of unknown variables in this Master's thesis and also to trust that my own decisions often were quite good solutions to the challenges. I will also like to thank the Radix community for helping with the more concrete challenges. The support from the Discord channel has been invaluable. On a more personal level, I would like to thank "Søsken Kollektivet" for always being there and for the consistent Sunday breakfasts throughout the semester.

Table of Contents

Acronyms

BTC, ETH, USDT, USDC Popular cryptocurrencies

Dapp Decentralized application

DeFi Decentralized Finance

DEX Decentralized Exchange

DLT Distributed Ledger Technology

LTV Loan To Value

NFT Non-Fungible Token

PoS Proof of Stake

PoW Proof of Work

SDK Software Development Kit

TVL Total Value Locked

UI User Interface

Glossary

Arbitrage	A strategy that uses price differences between markets in order to make a profit.
Blueprint	In Scrypto the concept of smart contracts is split into blueprints and components. If we want to compare it to object-oriented programming, blueprints are classes, and components are objects. A blueprint contains all variables that will be part of the state of each instantiated component. The blueprint also holds the code for methods used to update the state of the component.
Burn	Indefinitely removing tokens from circulation.
Composability	Composability in the context of DeFi is the property of being able to make use of multiple smart contracts in one transaction.
Flash loan	A loan type unique for DeFi. Allows a loaner to take up as much loan as there is liquidity in a pool, use it for whatever it wants, and pay back the loan in the same transaction. If the amount is not paid back within the same transactions, all actions are reverted.
Manifest	The way to build a transaction on Radix
Mint	Creating or adding new tokens to circulation.
Overcollateralized	A loan type where the value of the collateral exceeds the value of the loan.
Pool	A common term in DeFi that is often used for liquidity pools, lending pools, mining pools, etc. It refers to liquidity pools in this report. A liquidity pool is a collection of tokens locked in a smart contract to facilitate trades.
Sybil attack	An attack on a computer network where a user tries to take over the network by operating multiple identities and undermine the authority in the reputation system.
Token	A digital unit that represents an asset or a utility. For example, a cryptocurrency or an NFT.
Whitelist	A list with trustworthy addresses.

Introduction

The goal for this master's was to build further on the decentralized application (Dapp) that I started on in my specialization project [24]. Some weeks into the master's, I also found a clear milestone I would work towards; A hackathon arranged by RDX Works, where I ended up winning an honorable mention prize worth \$3,500. DefiFunds is a Dapp designed for both fund managers and everyday investors. This platform allows fund managers to create decentralized funds, subsequently earning a fee from investors. For everyday users, DefiFunds offers the convenience of investing in the decentralized finance ecosystem without the need for individual trading. There's no need for users to trust the platform or the fund managers to hold their funds, as they are securely held within a smart contract. The fund manager can trade with the funds, but it is not possible for anyone except the users to withdraw their share of the fund.

If you have a good understanding of blockchain and Ethereum, you can probably skip the background chapter on distributed ledger technology. However, I suggest taking a look at the subsection on the scalability problem to gain a better understanding of my motivation regarding building the Dapp on Radix instead of Ethereum. If you are familiar with Radix and developing on it as well, you can most likely skip the background chapter about Radix. When reading about the state of DefiFunds at the start of this master's thesis and the result chapter, I often refer to the appendix for code implementation; it could be wise to open a second pdf and read them side by side if you are reading this master's digitally.

1.1 Motivation

My main motivation for this master's comes from my desire to build the Dapp I started on in my specialization project to production. Motivation for my specialization project comes from my strong interest in distributed ledger technologies since 2018. The rationale behind choosing Radix as the platform for my Dapp is outlined in the next two paragraphs, while the third following paragraph explains why I wanted to build this Dapp specifically.

I have been using the Ethereum network for several years, and have seen its weaknesses and advantages. In my opinion, the main weakness with Ethereum is the scalability problem. It can only handle 30 transactions per second on average. The fees paid for a single transaction have sometimes been in the hundreds of dollars because of the scalability problem and the competitive fee structure on Ethereum. Radix, on the other hand, looks promising with its potential for linear scalability without losing composability.

Another advantage with Radix is its programming language, Scrypto, which is specifically designed for writing smart contracts on Radix. Scrypto is built on Rust and is often referred to as an "asset-oriented" language. It focuses on safe and efficient moving of resources. On the other hand, developing on Ethereum is mostly done with an object-oriented language called Solidity. As a result, resources are not a native element in the language, leading to a more complex logic.

When I started building the proof of concept for this Dapp, there was no competition for this kind of Dapp. There still aren't any competitors, which gives me extra motivation to build this Dapp to production since it actually can be useful for people. I often get questions from acquaintances about what cryptocurrencies they should buy and when they should buy or sell them. It would have been a lot easier to say: "buy this fund and let your money stay there for some years," instead of telling them what to buy and when. Most of my acquaintances only invest in funds, not single stocks or cryptos. This can be the perfect Dapp to introduce them to crypto.

1.2 Disclosure - Utilizing AI tools and text from my specialization project

In writing my master's thesis, I have utilized several AI tools, such as ChatGPT and Grammarly, to help me with the writing process. Additionally, I have incorporated text from my previous specialization project, which can be found here [24].

My supervisor has encouraged me to focus on producing the best possible master's thesis using the tools and resources available to me, even if that means reusing text from my previous project. Some sections of my master's thesis are kept in their original form, and others have been modified or added to, resulting in instances where the text in my master's thesis is the same or similar to the text that appeared in my specialization project. Instead of referencing to my specialization project every place I reuse the text, I have decided to include an overview of the reused text at the end of this section.

I was advised by my supervisor to use any AI tool that I liked to help me produce text, as long as it did not compromise the quality of the text. In some places, I have used text generated from ChatGPT by asking a series of follow-up questions and modified the AI-generated text where I felt it did not adequately explain the topic. I have also used it a lot for rewriting bad sentences.

By utilizing AI tools and incorporating relevant text from my specialization project, I am confident that my master's thesis will provide a thorough and thoughtful exploration of the topics at hand, while also showcasing my ability to work efficiently with the tools available to me.

Overview of text reuse from my specialization project

- Glossary and acronyms are mostly the same.
- Parts of the introduction and the motivation are inspired by my specialization project, since the master's thesis is structured similarly and the motivation is mostly the same.
- The background chapter about distributed ledger technology is a direct copy of my specialization project.
- About two-thirds of the "Radix and the Radix engine" section in the "Developing on Radix" chapter is a direct copy.
- The Scrypto section in the "Developing on Radix" chapter is mostly a direct copy, apart from the new subsection about blueprints. Images in this section are updated to a newer version of Scrypto.
- The blueprints section in the chapter about the state of DefiFunds is close to a direct copy of my specialization project's design and implementation section, but has some changes, so it fits well as a background chapter for this master's thesis instead of a result chapter.
- The possible tasks that haven't been solved in the future work section in the conclusion chapter are a direct copy, as they still are equally relevant.

Background: Distributed ledger technology

This chapter is written to give general knowledge about distributed ledger technology (DLT). Section 2.1 explains what a DLT is and introduces some examples of existing DLTs. Section 2.2 explains what a smart contract is and some associated challenges. Section 2.3 gives some examples of what these smart contracts are used for in decentralized finance (DeFi) and introduces a problem currently affecting the adoption rate of DeFi. Both Section 2.2 and Section 2.3 are written mostly with Ethereum in mind since Ethereum is the largest DLT with substantial smart contract functionality. Terms more specifically used for Radix are described in chapter 3.

2.1 Distributed ledger technology

Distributed ledger technology has become quite popular in the recent years. Primarily because of the technology later named blockchain, which was introduced in 2008 [36] as a part of a proposal for Bitcoin. Blockchain has become a relatively known technology in recent years and is often misused as the word for DLT. Other structures used as a DLT for cryptocurrencies are for example a directed acyclic graph; Iota is one of them[3]. There also exist other types of DLTs, and the Radix network which is going to be the main focus of this project is one of them.

2.1.1 Database

Storing and managing data is an important part of most applications today, and a database is an important tool for making that happen. A database is a collection of structured data that is often stored electronically. It is often modeled in rows and columns in a series of tables and is used for having easy access to data. The type of operations that can be performed on a database varies depending on the database type, and some basic examples of operations are, create, read, update, and delete. Three types of databases relevant for understanding a DLT are centralized, decentralized, and distributed databases [39].

A centralized database is stored in a single location and maintained by a single node. The main advantages of a centralized database are that it is easy to access, easy to coordinate data, and cheap to maintain since all data is stored in the same place. However, a centralized database has drawbacks in performance, redundancy, and availability. If the system is down for a while or fails totally, there will be no access to the data, and the data itself can be destroyed.

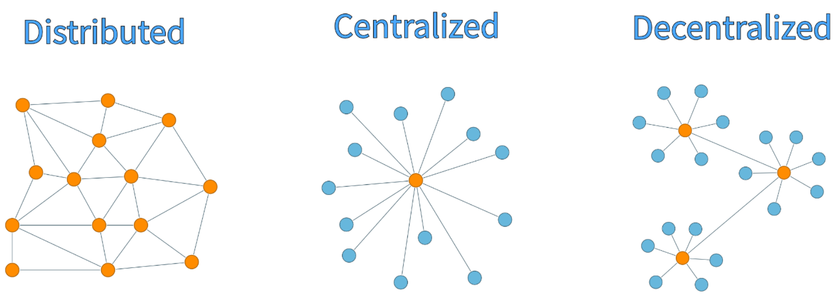


Figure 2.1: Visualizations of different database structures

Decentralized databases do not have central storage. The entire data is spread across multiple servers, often in different physical locations, and managed by several nodes. When a node modifies the data, it is reflected on all the other nodes. In contrast to a centralized database, decentralized databases are more redundant, but it is a bit harder to coordinate the data. Nodes are organized in a hierarchical structure where a set of nodes communicate with a specific set of nodes that contains all the data. A decentralized database can be seen as several smaller centralized databases where a superordinate set of nodes coordinate on having the same data.

A distributed database is similar to a decentralized one in that it also is stored in different physical locations. A distributed database is usually not organized in a hierarchical structure and often forms a mesh, as illustrated in Figure 2.1. All nodes have equal rights for changing the data in the network, which can lead to different nodes having different records of the database. Even though a distributed database is architecturally distributed, it is considered logically centralized, which means that operations performed on a node should lead to the same results independent of which node the operations were performed on. The process of reaching consensus on a distributed system is generally harder on a distributed system than on a decentralized one, since all nodes are part of the data storing. To reach consensus across the network, a consensus mechanism is used which will be more detailed explained in Section 2.1.3. A huge advantage of a distributed system is that it is far more reliable. If one node goes down, the network will still be able to communicate; the traffic will just go through other nodes. Since nodes can work independently, the network itself is generally more scalable and efficient.

2.1.2 Distributed ledger

A distributed ledger is a special type of distributed database [39], where only append and read operations are valid. If a person would want to change a value in the ledger, an append operation is done, and the old value is stored on the ledger. A ledger keeps track of all updates and stores them chronologically. A ledger is also tamperproof so that everyone can cryptographically check that the data has not been tampered with. Ledgers presume the existence of malicious nodes and use a consensus mechanism to achieve consistency across the network. DLT allows distributed ledgers to be run on arbitrary nodes where providers do not need to be trusted or known. The nodes can also arbitrarily join or leave the network without affecting consistency.

2.1.3 Consensus mechanism

To ensure that all nodes agree on a single database record, a consensus mechanism is used [12]. It is a crucial part of distributed ledger technology since no central authority can control the network or the data stored in it. For simplicity's sake proof of work (PoW) and proof of stake (PoS) are often referred to as consensus mechanisms, but in reality, it is only a part of the consensus mechanism.

PoW and PoS are the most common Sybil resistance mechanisms used in distributed ledger technology [12]. A Sybil attack is an attack where a user tries to take over a network by pretending to be many users or nodes. Since a DLT is a decentralized network with no input from a central authority, Sybil resistance is essential. In PoW a Sybil attack is minimized by getting users to spend a lot of CPU power and in PoS by letting users put their tokens up as collateral. A user will then have a weighted vote based on their CPU power used or collateral put up.

In this subsection, three different consensus mechanism is introduced; A PoW-based consensus mechanism often named Nakamoto consensus, a PoS-based consensus mechanism called Gasper and Cerberus which also uses PoS for Sybil resistance.

Nakamoto consensus - Proof of work based

Proof of work as a concept was introduced in 1993 by Cynthia Dwork and Moni Naor, and later applied by Satoshi Nakamoto in 2009 when Bitcoin was invented [29]. Nakamoto also came out with a whitepaper in 2008 where PoW and the technology in general are explained [36]. PoW is a cryptographic proof used to show that a certain amount of work has been done. On the Bitcoin network, PoW involves hashing nonces using the sha256 algorithm, searching for a hashed value with the required 0 bits. The average work required is exponential to the required zero bits and is dynamically adjusted based on the total work of all computers in the network. Once a hashed value with the required zero bits has been found, a block gets added to the blockchain, and it can't be changed without doing all the work again. If someone should want to change a block from the past, he needs to redo the work for all the blocks after it.

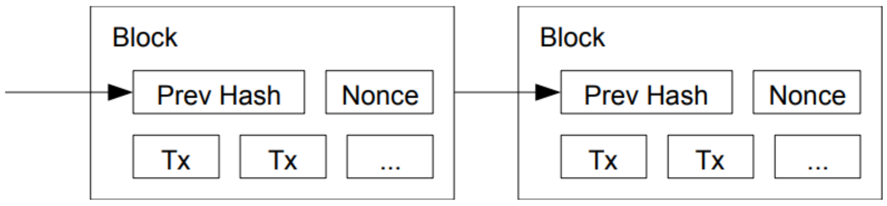


Figure 2.2: Chained blocks [36]

Another part of the consensus mechanism in Nakamoto Consensus is the longest-chain rule. The longest blockchain is considered the valid blockchain and is statistically going to be the chain with the majority of the work on it. If honest nodes then control more than 50% of the CPU usage, it controls the blockchain. If an attacker with less than 50% of the CPU power tries to attack the network, the probability of the attacker catching up to honest nodes diminishes exponentially as new blocks get added to the blockchain.

After one node has added a block to the blockchain, other nodes will update their record to include that block as well. If two nodes find a new block almost simultaneously, different nodes will have different records of what block is added. They will also save the other block in a branch in case that chain gets longer. Nodes will continue to work from the block they first saw, but will switch over if a new block gets added to the other.

Mining is a term used to describe the process of creating new blocks on the blockchain. When someone finds the hashed value with the required 0 bits, a new block is created, and a reward for finding the value is given to the node that found it. This technology gives incentives for nodes to mine and keeps the networks secure. A disadvantage of this feature is that it competes on CPU and energy usage. This has led to an estimated energy consumption of 130 TWh as of October 2022, which is comparable to the electricity usage of Argentina [6].

Gasper - Proof of stake based

Proof of stake is an alternative Sybil resistance mechanism and has been used on Ethereum since September 2022 [38]. Before that, Ethereum also used the Nakamoto consensus. Instead of using CPU power as the voting mechanism, Ethereum is using staked ether. In other words, ether that is locked up as collateral. If a node votes the same as the majority, it will be rewarded, but if it votes against it, it will be punished. A node with voting rights on Ethereum is called a validator and needs to hold 32 Ether as collateral. The network picks a random validator to propose a block, and the other validators votes for or against this block to be added to the blockchain.

For incentivizing people to stake and secure the network, there is something called staking reward [28]. This reward is adjusted based on how much ether is staked and consists of transaction fees from users and issuance of new ether from the Ethereum network itself.

Cerberus

Cerberus is the consensus mechanism that will be used by the Radix network in its final form. Radix uses a form of PoS as the Sybil resistance mechanism, called delegated PoS [19]. In delegated PoS, the users do not participate in the voting but rather give the voting power to a validator or several validators it trusts. This makes it easier for people to stake and generally increases network security since more tokens are needed for a Sybil attack. On Ethereum, each staker needs to run their own validator to be able to stake. However, several centralized solutions exists that let people stake Ethereum through them.

The consensus part of Cerberus is a lot more complex than Gasper and Nakamoto Consensus and will not be explained here, but can be read in detail in the Cerberus whitepaper [9], or in simpler form in the infographic series [44]. The main advantage of Cerberus is that it achieves unlimited linear scalability while still preserving atomic composability. Ethereum also plans to increase its scalability with sharding, but it is up to debate how composable it gets [13].

2.2 Smart contracts

2.2.1 What is a smart contract?

A smart contract is a contract that is automatically executed when certain conditions are met, and is typically stored on top of a DLT. There exists smart contracts on Bitcoin, but only simpler ones, since Bitcoin is Turing incomplete. On the other hand, Ethereum is Turing complete and has been the platform with the most smart contract development on. [31]. One can divide a smart contract into four phases to better understand how it works: creation, deployment, execution, and completion [55].

The first step is the creation of the smart contract. In this step, the parties involved in the deal must agree on certain parameters. A simple example can be a trade between two cryptocurrencies. One receives x amount of dollars, and the other part receives y amount of ether. After the deal is agreed on, the code needs to be developed.

After the smart contract is created, it needs to be deployed on top of a DLT. Since a DLT is immutable, the smart contract can't be modified after the deployment. If parts of the contract need to be modified, the contract needs to be redeployed. The old contract will still exist on the DLT. The code is often already created and deployed, so normal users can normally skip this and the previous step.

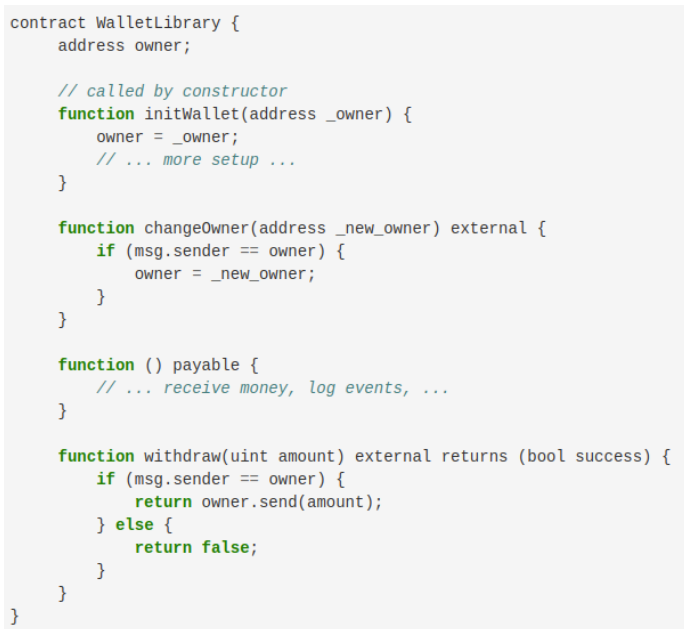
Once a smart contract is deployed on a DLT, all users of the network can call the contract. If all criterias for the smart contract are met, the smart contract executes. When a smart contract is called, transactions with all state changes defined by the smart contract are sent to the network and validated by the nodes on the network. A smart contract is also deterministic, meaning that the outcome of the contract will be the same for everyone who calls it.

After the transactions have been executed, new state changes according to the smart contract are updated. All transactions involved in the process and the new state changes are stored on the DLT. The cryptocurrencies from the trade have switched hands, and there was no need for a middleman.

Once a smart contract is deployed to a network, no third party controls the smart contract. Even though individuals or organizations create them, they have no exclusive right over them. An important part of a smart contract is that there is no need to trust the third party that developed the smart contract to use it. It can be verified by yourself or other parties you trust. Code is law, and there is no way of changing a smart contract once deployed.

2.2.2 Security in smart contracts

Even though smart contracts are secure in the way that no third parties can override them, there are still several security problems. The two main problems are that it's hard to write bug-free code and that it's hard to verify for the user what transaction he is signing. In a centralized banking system, the bank can change balances or other fields in a database if bugs should occur. In a decentralized system, it is not possible [55]. We will go through some examples to illustrate the problems.



```
contract WalletLibrary {
     address owner;

     // called by constructor
     function initWallet(address _owner) {
         owner = _owner;
         // ... more setup ...
     }

     function changeOwner(address _new_owner) external {
         if (msg.sender == owner) {
             owner = _new_owner;
         }
     }

     function () payable {
         // ... receive money, log events, ...
     }

     function withdraw(uint amount) external returns (bool success) {
         if (msg.sender == owner) {
             return owner.send(amount);
         } else {
             return false;
         }
     }
}
```

Figure 2.3: Bug in a smart contract [7]

Vulnerable wallet example

Figure 2.3 is a snippet of a simplified smart contract written in Solidity. It has a vulnerable bug and goes under the name "Party MulitSig Bug" [7]. This Particular bug was discovered by white hat hackers and drained around 78 million dollars worth of cryptocurrencies from the contract. The code snippet illustrated is a simplified version of the real contract, but is good for understanding the point. The first function sets the owner of the wallet and a lot of other parameters. The second function is a function that lets the current owner set a new owner. The fourth function is a function that lets the owner withdraw the funds. On Ethereum the default visibility for functions are public, so everyone can call them if not specified otherwise. The problem with the first function was that it was believed to only be callable once, but it could actually be called by everyone. Luckily a group of white hat hackers discovered this bug first, set themselves to owners, and withdrew all the funds.

Signing smart contracts

It may sound strange, but a common practice on Ethereum has become blind signing transactions. Blind signing means confirming transactions you do not know the full details of [34]. One of the reasons may be the quickly evolving space. The user experience has not kept up with the smart contract development and has led users needing to trust the smart contract instead of verifying it themself. When smart contracts first came, there was no way to go around blind signing if you wanted to use simple Dapps like Uniswap. Uniswap is the largest decentralized exchange by trading volume as of December 2022 [21]. The problem with many smart contracts is that the important contract details, like what amounts should be sent and received, can not be fully extracted from the contract in a user-readable language. The user interface can often display something like "data present" or a long hash value, instead of key details needed to verify what is signed.

Developers have in general focused on "ease of use", instead of secure implementations. When doing a token swap on Uniswap a user needs to confirm two transactions. The first transaction is a token allowance transaction that lets the smart contract spend the token. The other transaction is a transaction that executes the token swap. A common practice has been to give unlimited token allowance, meaning that users give the smart contract rights to spend an unlimited amount of that token on their behalf. In daily life, it can be compared to an automated bill from an electricity company. The company can charge you want they want. Still, they have little incentive to charge a higher amount than your bill was supposed to be. This will be categorized as illegal, and the bank can easily reverse the action for you [11]. Once a payment is made with smart contracts, the action is irreversible. Hackers have higher incentives here since is harder to get caught by governments, and the transaction can't be reversed by a third party.

Phishing

On a DLT, users have full responsibility for the funds themselves, which has led to lots of phishing scams. There are not that many reports on phishing scams related to smart contracts in general, but a report covering Non-Fungible Tokens (NFTs) shows that phishing scam is the most common scam type [5]. There are two main ways phishing attacks occur:

The first and most common attack type is through a fake popup that looks like a real custodial wallet, like Metamask. The provider typically asks users the re-enter their private key. Once entered the scammer has access to their wallet and can drain all NFTs and cryptocurrencies without any third party stopping the action.

The second most common attack type is encouraging users to sign malicious smart contracts. The main problem here is that blind signing is a common way of signing. Scammers typically make use of the unlimited token allowance function for fungible tokens or similar functions made for non-fungible tokens.

2.2.3 Oracle problem

Connecting real-world applications to a smart contract is quite challenging and is called the oracle problem [10]. A DLT only need to form consensus on binary data already stored on the DLT. The problem with connecting real-world data to a DLT is that it is hard to determine what data is correct. Take for example the price of ether. It can be one price on a specific exchange and another on a different exchange. The problem here is what source you should trust.

The main reason for wanting a smart contract is to achieve deterministic output based on the inputs. If the inputs can vary based on the different data sources, there is little use for a smart contract. Getting information from a trusted centralized third party would be more secure than getting information from a random data source. However, the trusted centralized data source still suffers from the same problems as centralized databases, like corruption and downtime. Smart contracts relying on an oracle are no more secure than the oracle itself.

Chainlink is the most used oracle service as of today [14]. It currently supports 14 platforms where some of them are Ethereum, BNB chain, and Solana. Chainlink has several features, and one of them is price feeds. It lets smart contracts retrieve the latest pricing data of a token. Each price feed is updated by multiple independent oracle operators. The number of oracles contributing to each price feed varies, and some price feeds are therefore more secure than others. The provided prices are validated and aggregated by a smart contract and form a final answer. For an update to take place, a minimum number of oracle nodes need to provide their price. The minimum number varies from price feed to price feed.

2.3 Decentralized finance

Decentralized finance, commonly known as DeFi, uses DLTs to manage financial transactions such as transferring, lending, borrowing, and trading. Today most of the financial transactions are done by centralized systems, but decentralized systems have started to take a little part of the financial sector. In this section, we will go through some examples of where DeFi is used today and some of its limitations.

2.3.1 Decentralized finance usecases

Self custody

Self-custody is the standard way to manage your assets in DeFi. Self-custody refers to users being in absolute control of their assets [37]. Their assets are backed by a private key and often stored in non-custodial wallets like Metamask or Ledger [35]. In traditional finance, it can be compared with holding your assets in cash or physical gold. DeFi's advantage regarding self-custody compared to centralized finance is that you still can use financial services, like sending money abroad, taking up a loan, and trading stocks.

In centralized finance, a third party controls your funds, usually a bank or an asset manager. If they were to go bankrupt, lots of your assets could get lost depending on your wealth and insurance. In 2013 users of the Bank of Cyprus lost up to 60% of their total holdings for deposits exceeding 100,000 EUR [39]. A bank can also choose to confiscate your funds or not let you send money to those you want. With DeFi and self-custody, these problems are eliminated as no third party or authority has access to your private key or funds.

The paradox with not having a third party to control your assets is that no one can access your funds if needed. If you lose your private key or password, there is no way to recover the funds. If you forget your password to the bank, you can likely go to the bank and ask for a new password.

Lending

In DeFi the two most common loan types are overcollateralized loans and flash loans [41]. The more commonly used in traditional finance, undercollateralized loan, is almost non-existent in DeFi due to handling defaults of loans. In traditional finance, an undercollateralized loan is dependent on a government and a debt collection agency to handle defaults of loans. In DeFi, there is currently no good implementation of this loan type, and we will focus on the two other loan types in this section. A flash loan can be fully executed on a DLT without any third party, or oracle. Overcollaterlized loans make use of an oracle, typically Chainlink, to get the price of different assets. Apart from that, overcollateralized loans do not need any oracle or third party.

An overcollateralized loan, is a loan where you put up more collateral than you take off loan. An example can be to put up 1000 dollars worth of ETH in collateral and take up a loan of 500 dollars worth of USDC. You now have a loan-to-value (LTV) ratio of 50% [4]. As the collateral fluctuates in price, the LTV ratio adjusts. If the LTV ratio gets to close to 100%, the risk of getting undercollateralized is there. To avoid this scenario, there is a liquidation threshold. When the liquidation threshold is met, any actor can liquidate you by buying part of your collateral at a discounted price.

A flash loan is a loan where you do not need to put up collateral, and you can take up as much loan as there is capital in the smart contract [4]. The loan is given and paid back within the same transaction. If a loan taker fails to pay back his loan within the same transaction plus interest, the loan is reverted. This is possible due to the atomic nature of smart contracts. Flash loans are not used in traditional finance, but play an important role in DeFi. We will explain how a flash loan works through a real arbitrage example that has happened on Ethereum.

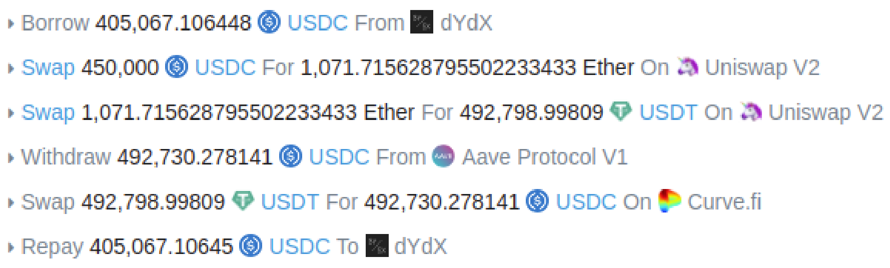

Borrow 405,067.106448 USDC From dYdX
Swap 450,000 USDC For 1,071.715628795502233433 Ether On Uniswap V2
Swap 1,071.715628795502233433 Ether For 492,798.99809 USDT On Uniswap V2
Withdraw 492,730.278141 USDC From Aave Protocol V1
Swap 492,798.99809 USDT For 492,730.278141 USDC On Curve.fi
Repay 405,067.10645 USDC To dYdX

Figure 2.4: An example of a flash loan used to do an arbitrage trade.
transactions hash: 0x01afae47b0c98731b5d20c776e58bd8ce5c2c89ed4bd3f8727fad3ebf32e9481 [1]

1. A user started by taking up a flash loan of 405k USDC (He already had 45k USDC).
2. He then exchanges 450k USDC for 493k USDT on Uniswap.
3. He exchanges 493k USDT for 493k USDC on Curve.
4. He repays the 405k of USDC and keeps the remaining 43k USDC in profit.

All these actions took place in the same transaction. If the borrower would not be able to pay back the USDC he borrowed, all the actions in the transaction would be reverted. An example on a reverted flash loan transaction can be found here: [2]. Flashloans have several other use cases too, like for example liquidating an overcollateralized loan that has reached its liquidation threshold.

Decentralized exchanges

Decentralized exchanges (DEXes) have had a rapid growth curve over the last few years. In January 2020, the monthly DEX volume was around 1 billion dollars, and two years later in January 2022 the monthly trading volume had risen to about 120 billion dollars [21]. A DEX is one of the better ways to show where smart contracts make a difference. Transaction costs are cut to a fraction, there are no restrictions on what cryptocurrencies can be traded, and they are available to use at any time.

There exist several types of decentralized exchanges. Some are order book based, like mostly done on centralized exchanges, while others use an approach with bonding curves. A simple but effective exchange method is the constant product automated market maker used by Uniswap [30]. Uniswap is the most popular DEX by trading volume as of December 2022 and has been it for a couple of years [21]. We will take a closer look at how Uniswap V2 works.

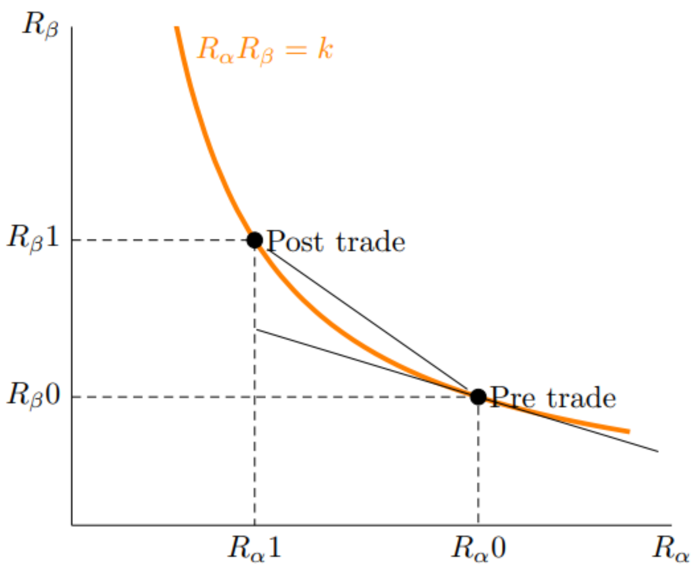


Figure 2.5: A constant product automated market maker [30]

A constant product automated market maker follows the simple function $R_\alpha R_\beta = k$. R_α is the reserve of asset α and R_β of the asset β. The product of them will be k before and after trades are done. When someone wants to do a trade, a transaction with x amount of α is sent to the smart contract containing the constant product function. The smart contract calculates the output amount of β based on the constant product function and sends the calculated amount of β in return. In an automated marker, the trades are made between market takers and liquidity providers. Market makers are buying or selling the asset, and liquidity providers provide liquidity to the pair. When providing liquidity to a pair, you send α and β to the smart contract in the same ratio as the total liquidity in the smart contract. To incentives people to provide liquidity to the smart contract, they receive a small fee from each trade.

2.3.2 Scalability problem

There has been, and still is, a problem with DLTs not being scalable enough to meet the high demand. The average transactions per second is about 15 on Ethereum as of 2022 [18]. Users of the Ethereum network compete to be among those 15 transactions by overpaying each other on fees, which has led to a very high cost for using the network. The average transaction cost for using Ethereum was in the range of 20-50$ in q4 2021 [17]. This has led to average users not being able to use the network. In response to the high price of using the network, an alternative DLT called Binance smart chain was made. It was able to handle a lot more transactions per second but met criticism for not being decentralized enough. It only has 21 nodes validating the network and was considered a centralized solution by many [32]. In this subsection, we will further look into the scalability problem on blockchains and possible solutions for it.

The Scalability Trilemma

Vitalik Buterin, the founder of Ethereum, states that there is a scalability trilemma where you can only get two out of three properties on a blockchain, if you stick to "simple" techniques. The three properties are scalable, decentralized, and secure [8]. The scalability trilemma is a hypothesis thought to be true by early developers, but hasn't any proof for or against it. The properties are defined as follows:

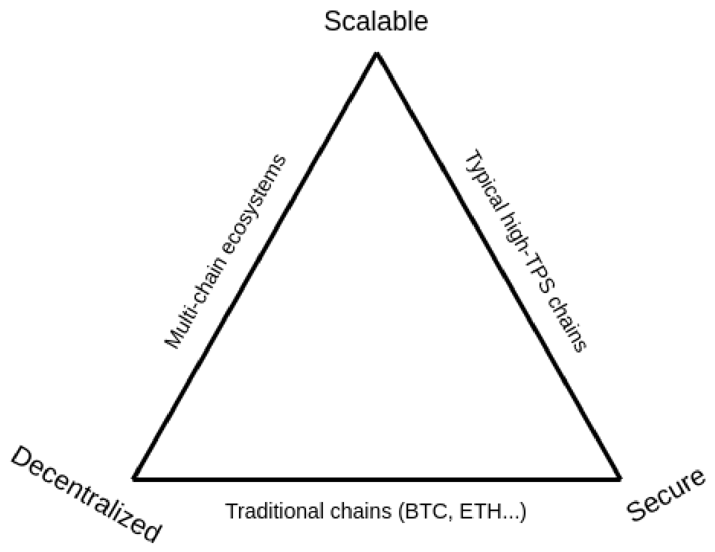


Figure 2.6: The Scalability Trilemma [8]

- Scalable: The blockchain can process more transactions than a single ordinary node. How scalable the blockchain is, is often measured in how many transactions it can handle per second.
- Decentralized: The blockchain can run without trusting a small group of centralized actors. The more nodes, and the less power they have individually, the more decentralized.
- Secure: The blockchain can resist a large number of malicious nodes participating in the consensus mechanism. Ideally, 49% of the nodes can act dishonestly, and the network still not be corrupt.

Sharding

Sharding is one of the solutions for making blockchains more scalable while still holding the decentralized and secure property from the Scalability trilemma. It is also the primary method protocol developers on Ethereum are working on. In simple terms, sharding means splitting up the verification process horizontally so that only a subset of all nodes verifies a block in the blockchain. This is done to spread the load so that the overall network can handle more transactions [8].

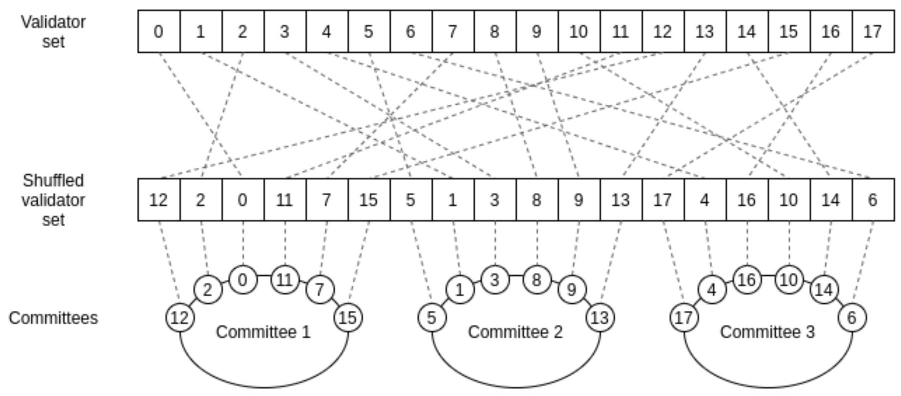


Figure 2.7: Sharding through random sampling [8]

We will go through random sampling, a simplified version of the sharding method that will be implemented on Ethereum [8]. See Figure 2.7. Say we have 18 validators and three blocks that need to be verified before the next set of blocks is incoming. We start by shuffling the validator list and randomly selecting a committee of validators to verify a block. Each validator in the committee verifies the block and publishes a signature as proof that they verified it. A validator only verifies the single block it's assigned. It also needs to verify the signature of the two other blocks. Verifying a signature is a lot faster than verifying a hole block. The reason for shuffling the validator list before each set of blocks is to prevent malicious actors from easily corrupting a committee. They now need to control close to a majority of all validators, instead of only a majority of the validators in a committee.

Composability

Sharding on Ethereum is thought to meet all three properties on the scalability trilemma, but it's up for debate how composable it gets after its sharding [13]. Composability in the context of DeFi is the property of being able to make use of multiple smart contracts in one transaction [20]. The reason that composability is important is that lots of Dapps make use of it today. The clearest example is the use of flash loans. Between the loan and the payback, the user makes use of several smart contracts. If those smart contracts were on different shards, the atomicity of the whole flash loan would not be possible, and if a flash loan can't be atomic, a flash loan can't exist.

On Ethereum, the composability question is still up for debate, but Radix claims to have solved the trilemma and the composability problem using its unique upcoming consensus mechanism called Cerberus [9]. The claims made by Radix are also verified in another independent paper published by the University of California Davis [27].

Background:
Developing on Radix

This chapter is written to give general knowledge about developing on Radix. Section 3.1 gives genereal knowledge about Radix. Section 3.2 explains the basics of the programming language used on Radix. Section 3.3 explains how sending a transaction on Radix works. Section 3.4 gives examples of what frontend tools are available and shows how these tools relate to the Dapp. The majority of the information in this chapter is taken from the Scrypto 101 course [49] and the Radix docs [53].

3.1 Radix and the Radix Engine

Radix is a DLT that is made to handle the problems that DeFi is currently facing. Radix claims to have solved the scalability trilemma without breaking atomic composability. It has also made its own language called Scrypto as an answer to the complex process of developing Dapps on DLTs like Ethereum.

The Radix Engine is the application layer on Radix and is similar to the Ethereum virtual machine on Ethereum. It is built to work hand in hand with Radix's future consensus layer called Cerberus. The main difference between The Radix Engine and the Ethereum virtual machine is that the Radix Engine is built around an asset-oriented approach, and uses a finite state machine to manage tokens. The Radix Engine makes sure that tokens never can be drained or lost and gives tokens a similar behavior as physical coins.

Currently, the Radix Olympia Mainnet is live, enabling basic functions such as token transfers and staking to secure the network. The Babylon Mainnet is estimated to go live on July 31, 2023, introducing smart contract functionality to the mainnet. At present, all smart contract features are only available on public test networks. This background chapter will primarily concentrate on the features set to be activated with the Babylon Mainnet launch.

3.2 Scrypto

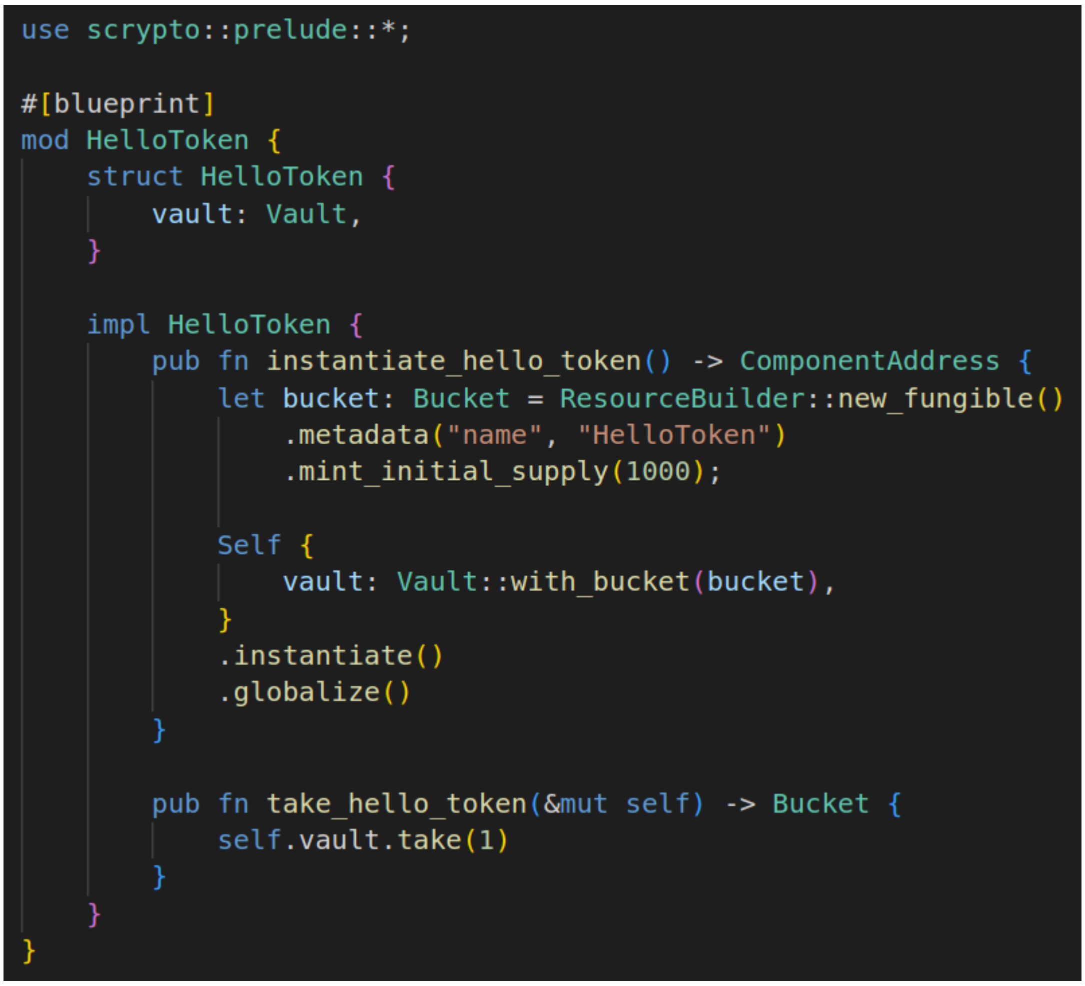
Scrypto is a programming language based on Rust. It is designed to develop Dapps with asset-oriented features and is the native language on Radix. In this section, we will take a look at important parts of the language, such as blueprints, resources, and authorization.

3.2.1 Blueprint

In Scrypto, the concept of smart contracts is divided into blueprints and components. Drawing a comparison to object-oriented programming, blueprints are similar to classes, while components resemble objects. A blueprint serves as a template for components. It contains functions, methods, and a declaration of the state structure. Components are individual instances of blueprints and possess an internal state. They also have the ability to hold resources.

Hello Token Example

Scrypto is an asset-oriented language, and the traditional "Hello World" example is replaced with "Hello Token". The simplest way to illustrate how Scrypto works is by creating a blueprint with a function that instantiates a component with some tokens and a method that takes some tokens from that component.



```
use scrypto::prelude::*;

#[blueprint]
mod HelloToken {
    struct HelloToken {
        vault: Vault,
    }

    impl HelloToken {
        pub fn instantiate_hello_token() -> ComponentAddress {
            let bucket: Bucket = ResourceBuilder::new_fungible()
                .metadata("name", "HelloToken")
                .mint_initial_supply(1000);

            Self {
                vault: Vault::with_bucket(bucket),
            }
            .instantiate()
            .globalize()
        }

        pub fn take_hello_token(&mut self) -> Bucket {
            self.vault.take(1)
        }
    }
}
```

Figure 3.1: HelloToken - A simple example of a Blueprint

In this example, the HelloToken blueprint consists of a struct and an implementation block. The struct defines the internal state of the component, which in this case is a vault that will hold the tokens. In a general blueprint, the struct can hold several different types, such as decimal, vector, etc. The implementation block contains a function and a method. The difference between a function and a method in Scrypto is that functions do not depend on an internal state, while methods require a reference to self so that they can read and modify the internal state. A meaningful blueprint will at least contain a function and a method, which is the case for this HelloToken example:

instantiate_hello_token: This function creates an instance of the HelloToken struct. It first creates a bucket with 1000 tokens, then initializes the vault with that bucket. The struct is instantiated using the .instantiate() method, and the resulting component address is made available globally with .globalize(). The .instatiate() method transforms the Rust value into a component, and the .globalize() method makes the component globally accessible by giving the component a component address.

take_hello_token: This method allows a user to take one token from the vault. It takes a mutable reference to self, allowing it to modify the component's internal state.

3.2.2 Resources

On the Ethereum network, tokens are implemented by developers as smart contracts. Two of the most common token standards on Ethereum are ERC-20 tokens which are a standard for fungible tokens, and ERC-721 which is a standard for non-fungible tokens [40]. The token standard has its own set of rules and methods related to it. Sending a token on the Ethereum network actually means calling a method on a smart contract that someone deployed. A token contract is simply a mapping of balances to addresses.

On the Radix network tokens, also referred to as resources, are natively understood by the platform. The whole Scrypto language is designed around handling resources in a safe way and is taken care of by the Radix Engine itself. The developer can then think of a token transfer as actually sending a token.

Figure 3.2 is an example of how a token is defined in Scrypto. It contains flags telling what metadata is connected to it, if it is burnable and mintable, what the initial supply is, and how many subparts a token can be divided into. The "rule!" function defines who can mint and burn the token and will be explained more about in Section 3.2.3.

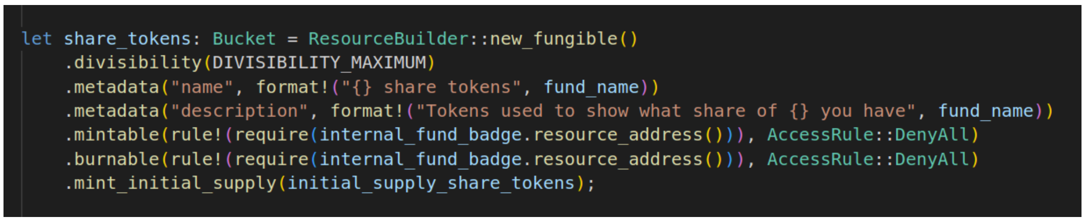

```
let share_tokens: Bucket = ResourceBuilder::new_fungible()
    .divisibility(DIVISIBILITY_MAXIMUM)
    .metadata("name", format!("{} share tokens", fund_name))
    .metadata("description", format!("Tokens used to show what share of {} you have", fund_name))
    .mintable(rule!(require(internal_fund_badge.resource_address())), AccessRule::DenyAll)
    .burnable(rule!(require(internal_fund_badge.resource_address())), AccessRule::DenyAll)
    .mint_initial_supply(initial_supply_share_tokens);
```

Figure 3.2: A resource defined in scrypto

Resource container

Buckets and vaults are the two resource containers that exists on the Radix Engine. They are relatively similar and both of them can only hold one type of resource. The main difference between buckets and vaults is that buckets are transient. That simply means that buckets only exists within the length of the transaction itself. Vaults on the other hand are permanent resource containers. When a transaction has finished, the resources that still are in buckets need to be put in a permanent resource container, or else the transaction will fail.

Finite State Machine

Resources in Radix are implemented through a finite state machine; see Figure 3.3. When a resource is first minted, it is put into a bucket. A resource can then be moved to a vault and at a later stage taken from that vault. A resource always needs to be in a bucket or a vault, and it can never be in a bucket and a vault at the same time. In this model, we can't have dangling resources. So if a resource is not put into a bucket or a vault, it needs to be burned. The finite state machine ensures that these principles are followed and returns an error if a developer tries to do otherwise.

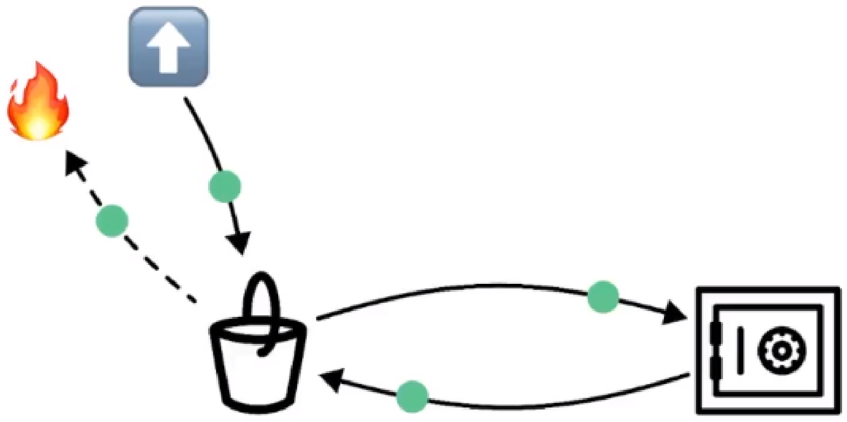


Figure 3.3: Finite State Machine [49]

The reason for having a finite state machine is to make tokens intuitive to understand and easier to avoid critical bugs when developing. Resources are more reliable, and states which are not thought about yet are impossible. Finally, this makes it safer and faster to develop new Dapps.

3.2.3 Authorization

In a smart contract, all methods are permissionless by default. This means that by default everyone can call a public method on a component. To restrict this access, some kind of authorization is needed. Scrypto's authorization model starts with three core concepts: badges, proofs, and access rules.

Badge

A badge is a resource used to gain access to restricted methods or actions. It is not a special type of resource; it is just a regular resource and is considered as a badge if it is used for authorization. On other networks such as Ethereum, the normal way to do authorization is through an address-based model. When you then want to restrict access to a method you verify that the caller's address is equal to the address needed to call that method. On Radix, a badge is used for authorization. If you would like to change an admin you can simply send that badge to the new admin. The idea of a badge is designed around the assets-oriented approach and is a more flexible way to restrict access.

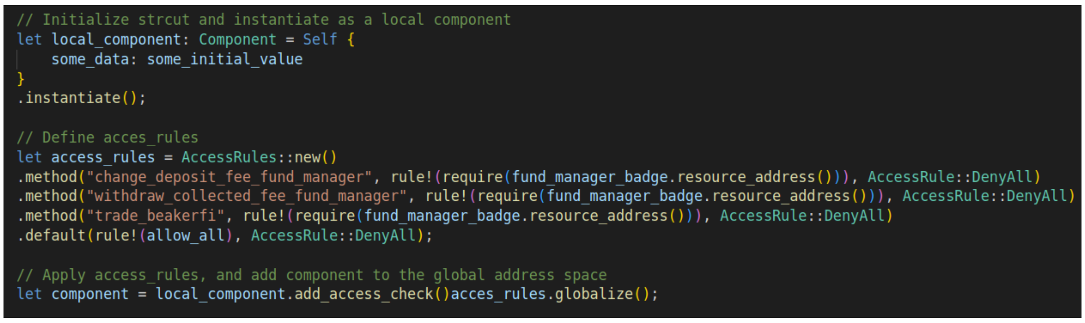
Proof

When you want to show that you own a specific badge, you create a proof of the badge and send the proof. Proofs are a way to show that you own a specific badge without sending the badge itself. If you send the badge instead of the proof, it is the same as sending over the access rights. Proofs are a copy of the badge and only exists within the duration of a transaction. A proof can be created by all types of resources and can be used to show that you own specific tokens.

Access Rules

Access rules can be set on several actions on resources like shown in Figure 3.2. Here a specific badge is required to mint or burn the resource. There exist several other access rules, and some of them are "allow_all", "deny_all", "require_any_of", and "require_amount". You can also combine several of these access rules together by using logical operators.

Access rules can also restrict access to certain methods on a component. By default, all public methods are accessible by everyone, but you can define access rules for public methods on a component like shown in Figure 3.4. To be able to call a method with an access rule you need to show a proof of the required badge.



```
// Initialize strcut and instantiate as a local component
let local_component: Component = Self {
    some_data: some_initial_value
}
.instantiate();

// Define acces_rules
let access_rules = AccessRules::new()
.method("change_deposit_fee_fund_manager", rule!(require(fund_manager_badge.resource_address())), AccessRule::DenyAll)
.method("withdraw_collected_fee_fund_manager", rule!(require(fund_manager_badge.resource_address())), AccessRule::DenyAll)
.method("trade_beakerfi", rule!(require(fund_manager_badge.resource_address())), AccessRule::DenyAll)
.default(rule!(allow_all), AccessRule::DenyAll);

// Apply access_rules, and add component to the global address space
let component = local_component.add_access_check()acces_rules.globalize();
```

Figure 3.4: Access rules

3.3 Transaction Manifest

Radix introduces a new way of sending transactions with the help of a transaction manifest. It will become the standard way of sending transactions on Radix with the Babylon release. Transaction manifests facilitates the movement of resources between components. It is achieved through a sequence of instructions using a specific instruction set designed for transaction manifests. The Radix Engine processes these instructions in order, and if any step fails for any reason, the entire transaction fails, and none of the steps are committed to the ledger. This ensures the atomic nature of the transactions.

Transaction manifests are human-readable, making it easy for developers to let users understand what they are signing. When ready to submit, the transaction manifest is translated into a binary representation and cryptographically signed, creating a final transaction that can be sent to the network and processed by the Radix Engine. The transaction manifest is a solution to the problem with blinding signing explained in Section 2.2.2.

In this section, I will introduce two new concepts, "The Worktop" and "The Authorization Zone". I will also explain how fee payment work and come with some concrete examples to better understand how the transaction manifest works.

The Worktop

The worktop is a resource management area for a transaction during execution. Resources returned from component calls are automatically placed on the worktop. From there, the manifest may specify that resources on the worktop should be put into buckets, which can be passed as arguments to other component calls.

Additionally, the manifest may have ASSERT commands. These commands examine the resources on the worktop, and if a certain resource is found to be lacking, the entire transaction will be reverted. This is useful to guarantee the results of a transaction even if the exact outcome of a component is uncertain.

The Authorization Zone

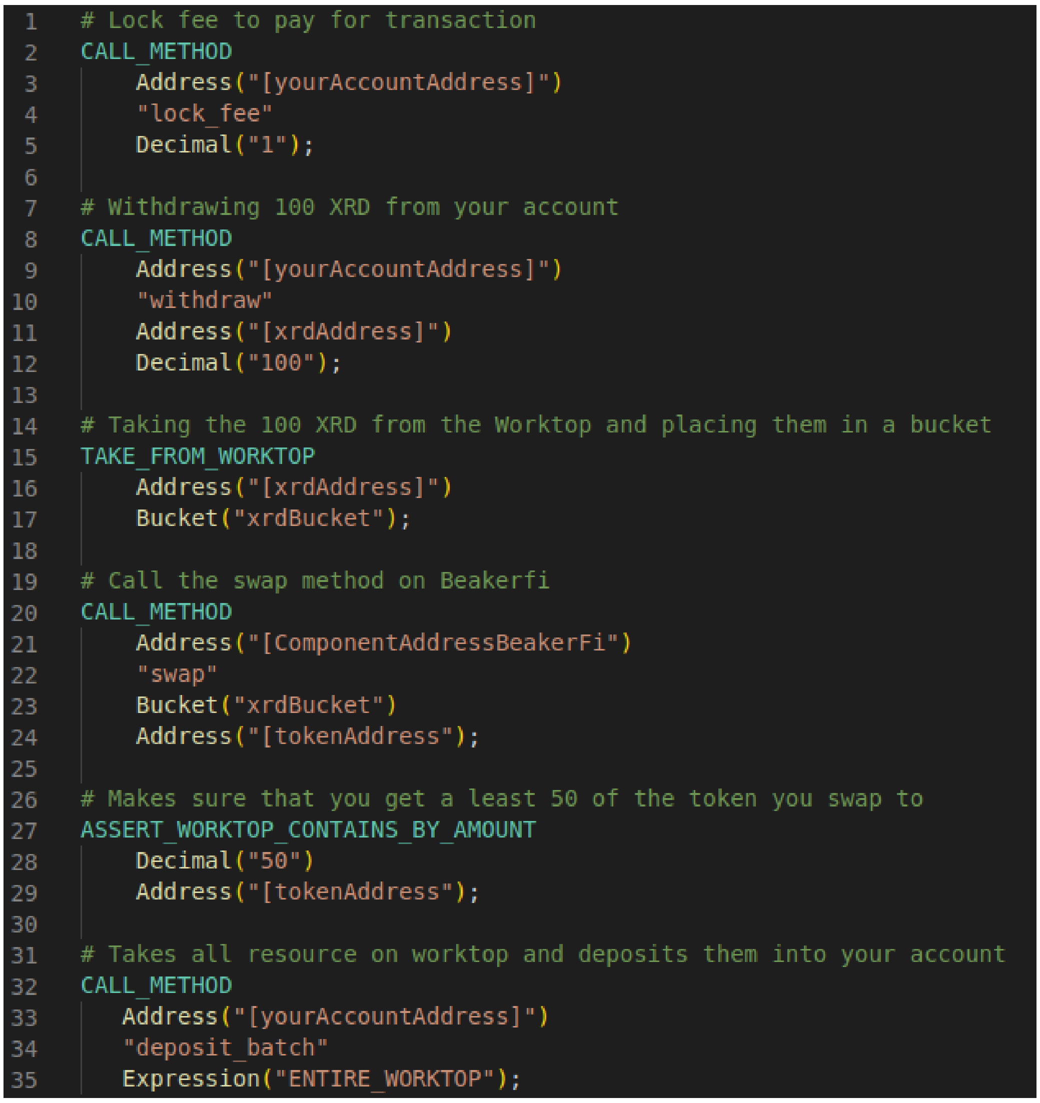
The authorization zone is another unique feature of the transaction layer, which is used specifically for authorization. Instead of holding resources, the authorization zone holds proofs. A proof is a unique item that verifies the ownership of a particular resource or set of resources. When a method is called by the transaction manifest, the proofs present in that transaction's authorization zone are automatically compared to rules governing access to that method. If the rules aren't met, the whole transaction will revert.

Fee payment

To run a transaction on the Radix network, you have to pay a fee. The fee amount reflects the burden the transaction puts on the network. On Ethereum, the one who executes the transaction needs to pay for the transaction to go through, but on Radix, you can specify otherwise in the transaction manifest. Most commonly, it is the user account that pays for the fee on Radix as well, but a vault in a Dapp can also be specified to pay for the transaction. The fee can also be paid by multiple vaults.

3.3.1 Example - A simple swap

We will look into a simple example on how to use the transaction manifest. We will make a swap using BeakerFi, which is a DEX on Radix.



```
1   # Lock fee to pay for transaction
2   CALL_METHOD
3       Address("[yourAccountAddress]")
4       "lock_fee"
5       Decimal("1");
6
7   # Withdrawing 100 XRD from your account
8   CALL_METHOD
9       Address("[yourAccountAddress]")
10      "withdraw"
11      Address("[xrdAddress]")
12      Decimal("100");
13
14  # Taking the 100 XRD from the Worktop and placing them in a bucket
15  TAKE_FROM_WORKTOP
16      Address("[xrdAddress]")
17      Bucket("xrdBucket");
18
19  # Call the swap method on Beakerfi
20  CALL_METHOD
21      Address("[ComponentAddressBeakerFi")
22      "swap"
23      Bucket("xrdBucket")
24      Address("[tokenAddress");
25
26  # Makes sure that you get a least 50 of the token you swap to
27  ASSERT_WORKTOP_CONTAINS_BY_AMOUNT
28      Decimal("50")
29      Address("[tokenAddress");
30
31  # Takes all resource on worktop and deposits them into your account
32  CALL_METHOD
33     Address("[yourAccountAddress]")
34     "deposit_batch"
35     Expression("ENTIRE_WORKTOP");
```

Figure 3.5: A simple swap - Transaction manifest

Lines 2-5: Used to specify the fee required to pay for the transaction.

Lines 8-12: We withdraw some XRD from the caller account, since we need them to make the swap.

Lines 15-17: Calling the method above will place the Radix tokens on the worktop. We take them from the worktop a put them into a bucket so we can move them around.

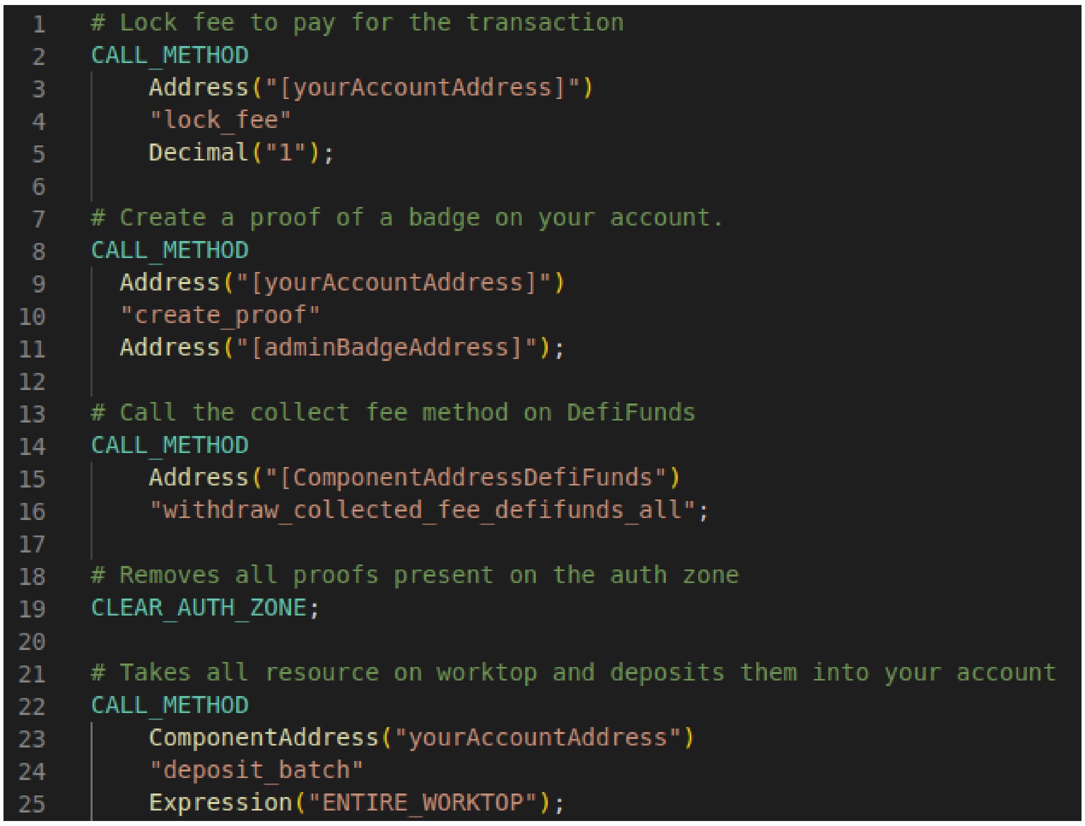
Lines 20-23: We call the swap method on BeakrFi. The swap method takes a bucket and an address as arguments. The bucket holds the tokens that are going to be swapped, and the address is the token that you want to swap them into.

Lines 26-28: When we use the swap method, we are expected to receive some tokens in return. We can add an assert function to make sure that the user receives at least 50 units of that token.

Lines 31-34: We deposit the tokens on the worktop, received from the swapping method, into our account.

3.3.2 Example - Protected method

We will take a look at how we use proofs and the authorization zone in a transaction manifest to call methods protected by access rules.



```
 1  # Lock fee to pay for the transaction
 2  CALL_METHOD
 3      Address("[yourAccountAddress]")
 4      "lock_fee"
 5      Decimal("1");
 6
 7  # Create a proof of a badge on your account.
 8  CALL_METHOD
 9    Address("[yourAccountAddress]")
10    "create_proof"
11    Address("[adminBadgeAddress]");
12
13  # Call the collect fee method on DefiFunds
14  CALL_METHOD
15      Address("[ComponentAddressDefiFunds")
16      "withdraw_collected_fee_defifunds_all";
17
18  # Removes all proofs present on the auth zone
19  CLEAR_AUTH_ZONE;
20
21  # Takes all resource on worktop and deposits them into your account
22  CALL_METHOD
23      ComponentAddress("yourAccountAddress")
24      "deposit_batch"
25      Expression("ENTIRE_WORKTOP");
```

Figure 3.6: Protected method - Transaction manifest

Lines 2-5: The first lines in a transaction manifest is used to specify the fee required to pay for the transaction.

Lines 8-11: We create a proof from the admin badge we hold, to prove that we are the admin. The method will fail if a person tries to call this method without holding the badge.

Lines 14-16: Calling the method above will place the proof in the authorization zone. The withdraw method is protected by an access rule and requires the admin badge to be in the authorization zone. The collected fee is then withdrawn to the worktop.

Line 19: Removes all the proofs on the authorization zone, since no proofs can be in the authorization zone at the end of a transaction.

Lines 22-25: We deposit the tokens on the worktop, received from the collect fee method, into our account.

3.4 Frontend integration

3.4.1 Radix Wallet

Users of a Dapp on Radix will interact with the smart contract of the Dapp through the help of the Radix wallet. The Radix wallet is a mobile app built by RDX Works and offers a secure and convenient way to manage accounts, send transactions, and sign smart contracts.

Currently, there is only a mobile wallet for developer purposes, which can be downloaded through TestFlight. A Chrome extension in developer mode will also need to be downloaded. The Chrome extension is not a wallet itself; it is simply an agent that facilitates a connection between the frontend of a Dapp and the mobile wallet. You can find instructions on how to download it in the Radix docs [51].

On Ethereum, the most popular wallet is Metamask. It is available in the browser and as a mobile app, and lets users sign transactions on Ethereum and other similar networks [33]. One of the main problems with this wallet is that users often need to blind-sign transactions, which is explained more about in Figure 2.2.2. One of the advantages with the Radix wallet, is that you can see what you are signing with the help of the transaction manifest. The transaction manifest is shown directly in the Radix wallet designed for developers, but it is planned to be more human-readable in the app designed for users. For the unknowing, it may seem strange that Metamask hasn't made the user experience better by letting users see what they are signing. The reason is that it is a lot harder to do on Ethereum than on Radix. The problem with blind singing was thought better through when Radix was designed.

3.4.2 Radix Dapp Toolkit

The Radix Dapp toolkit is a developer toolkit made by RDX Works that consists of three main components and makes it easier for developers to connect users and their Radix Wallet to their Dapps.

- **Connect Button** - A consistent Radix branded user interface (UI) element that works together with the wallet SDK, and lets users see the current status of the connection between the mobile wallet and the Dapp.
- **Wallet SDK** - A software development kit (SDK) that facilitates communication with the Radix wallet. It lets you send transactions to the wallet and request various forms of data.
- **Gateway SDK** - A thin wrapper around the Babylon gateway API.

Connect Button

Figure 3.7 shows the connect button. In an app, you normally log in with a username and a password, but on Radix you log in using your wallet. This will be the standard login process for every Dapp connecting to the Radix network. Your wallet can have multiple accounts, each with its corresponding account address. The state of the account address can be looked up using the gateway API, and all tokens and transactions are also publicly visible on the ledger.

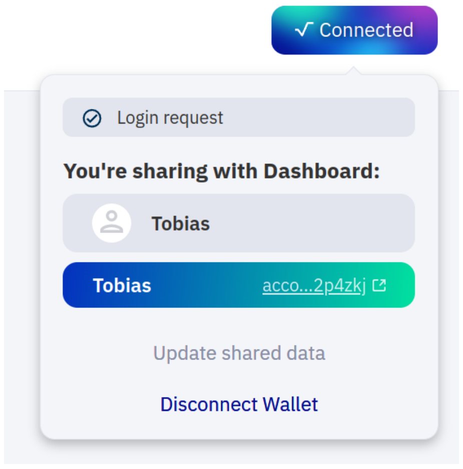
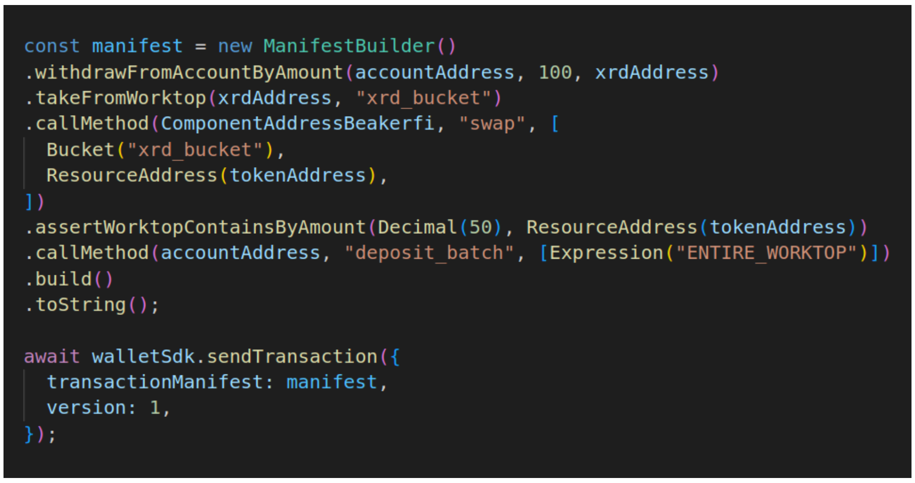


Figure 3.7: Logged into a Dapp

Wallet SDK

The wallet SDK contains several functions, but here I will only show you how to send a transaction to the Radix network. A detailed description of the Wallet SDK can be found on GitHub [52].

To send a transaction using the wallet SDK you first need to create a transaction manifest using the manifest builder. The transaction manifest shown in Figure 3.8 is almost the same as in Figure 3.5. The difference here is that "lock_fee" and "assert_worktop" are excluded. This is because the manifest in Figure 3.8 is a "manifest stub" and not a complete manifest. The Radix wallet will automatically add the "lock_fee" method. The assert command, with the expected returns from the swap, is added by the user in the wallet. After you have created the "manifest stub," you simply call sendTransaction on walletSdk. In this example, walletSdk is already connected to the Dapp with the help of the connect button and a request login function from the wallet SDK.



```
const manifest = new ManifestBuilder()
.withdrawFromAccountByAmount(accountAddress, 100, xrdAddress)
.takeFromWorktop(xrdAddress, "xrd_bucket")
.callMethod(ComponentAddressBeakerfi, "swap", [
  Bucket("xrd_bucket"),
  ResourceAddress(tokenAddress),
])
.assertWorktopContainsByAmount(Decimal(50), ResourceAddress(tokenAddress))
.callMethod(accountAddress, "deposit_batch", [Expression("ENTIRE_WORKTOP")])
.build()
.toString();

await walletSdk.sendTransaction({
  transactionManifest: manifest,
  version: 1,
});
```

Figure 3.8: Send a transaction using Wallet SDK

Gateway SDK

The Gateway SDK is a thin wrapper around the gateway API, and a description of how to use it can be found on GitHub [46]. I will in the rest of this subsection explain the gateway API.

The gateway API is a high-level API intended to be used by Dapps and general network clients. The full API specification can be found in the API docs [48]. Common use cases are reading the state of the ledger and getting the status on a transaction. The API is divided into four sup-API:

- **Status** - This sub-API provides status and configuration details for the gateway.
- **Transaction** - This sub-API is intended for transaction construction, preview, submission, and monitoring the status of the individual transactions.
- **Stream** - This sub-API is dedicated to reading committed transactions from the network.
- **State** - This sub-API is used for reading the network's current or past ledger state.

To explain the practical application of the Gateway API, let's consider a straightforward example. Imagine you use a Dapp where you swap some USD for some XRD. You sign a transaction that swaps 10 USD for 100 XRD. After the transaction is signed and sent to the network, the transaction API is used to get the status of the transaction to see if it has succeeded, failed, or is pending. When the transaction has succeeded, the state API is used to get the new state on the ledger. The state on your account is now 10 USD less than it was and 100 XRD more than it was.

3.4.3 How the frontend tools relate to each other

The blue boxes in Figure 3.9 are frontend tools made by RDX Works, and the green boxes are made by the developer. I will explain how the different parts work together by describing the user flow of a simple DEX.

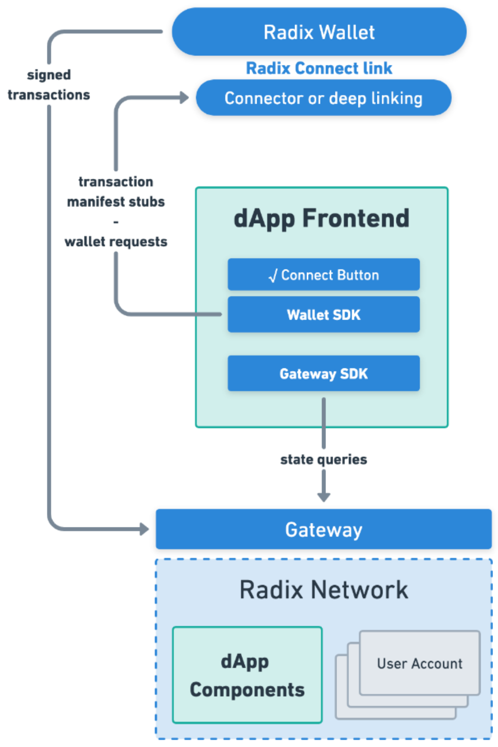


Figure 3.9: Shows the relation between the frontend tools and the Dapp [54].

1. The user loads the website. By loading up the website, some state queries are sent to the Radix network via a gateway to get the amount of tokens in the pools. The price of the token pairs is calculated based on those state queries, and the user sees the prices of the tokens.

2. The user clicks on the connect button on the website and logs into his account. The website updates its content based on that. For example, it will show how many tokens he has in his wallet. A state query with the address related to his account was sent to the gateway API.

3. The user will select what token he wants to swap to and what amount he will swap. He then presses the swap button. The website builds a transaction "manifest stub" based on these parameters with the help of the wallet SDK. The "manifest stub" is then sent to the Radix wallet.

4. The swap that is going to take place is shown to the user in his wallet. The wallet automatically sets the fee for the transaction. The user can also set parameters for the transaction, for example, that he wants a minimum of that token for the swap to be successful. The wallet completes the transaction manifest based on those parameters. Finally, the user is asked to sign the transaction, and the transaction is sent to the Radix network.

5. The website sends new state and status queries, and updates its content based on that. The amount of tokens in the user wallet and the price for the tokens are updated.

3.5 State of Radix during the project

The Radix public network has been live since July 2021 [45]. Simpler functionality like token transfers are possible, but all Scrypto features will first come to mainnet with the Babylon release, estimated to arrive on 31. July 2023. Babylon Betanet and Scrypto version 0.7 went live just after my specialization project ended, and it was also the version I started with when starting on this master's [42]. Babylon Betanet is a testnet built for developers to enable testing of applications on a live network. The Betanet was a replacement for the earlier Alphanet, and was also replaced by Babylon RCnet on 31.03.2023 [43]. Replacement of the network meant that the earlier version was shut down.

The Babylon Betanet enabled publishing blueprints to the network and sending transactions with function and method calls. You could also query state and other relevant info from the network by calling the gateway API. [48]. The Babylon Betanet enabled an easy way to test the application from the frontend using a mobile test wallet. The RCnet was an upgraded version of the Bayblon Betanet and enabled new features. Still, it also had braking changes, which made my application for the Babylon Betanet not to work, without updating it.

The Babylon testnets and Scrypto are under constant changes with several breaking changes. In addition, the betanet experienced ledger resets and downtime during my development. My problems related to the constant changes are discussed in chapter 6.

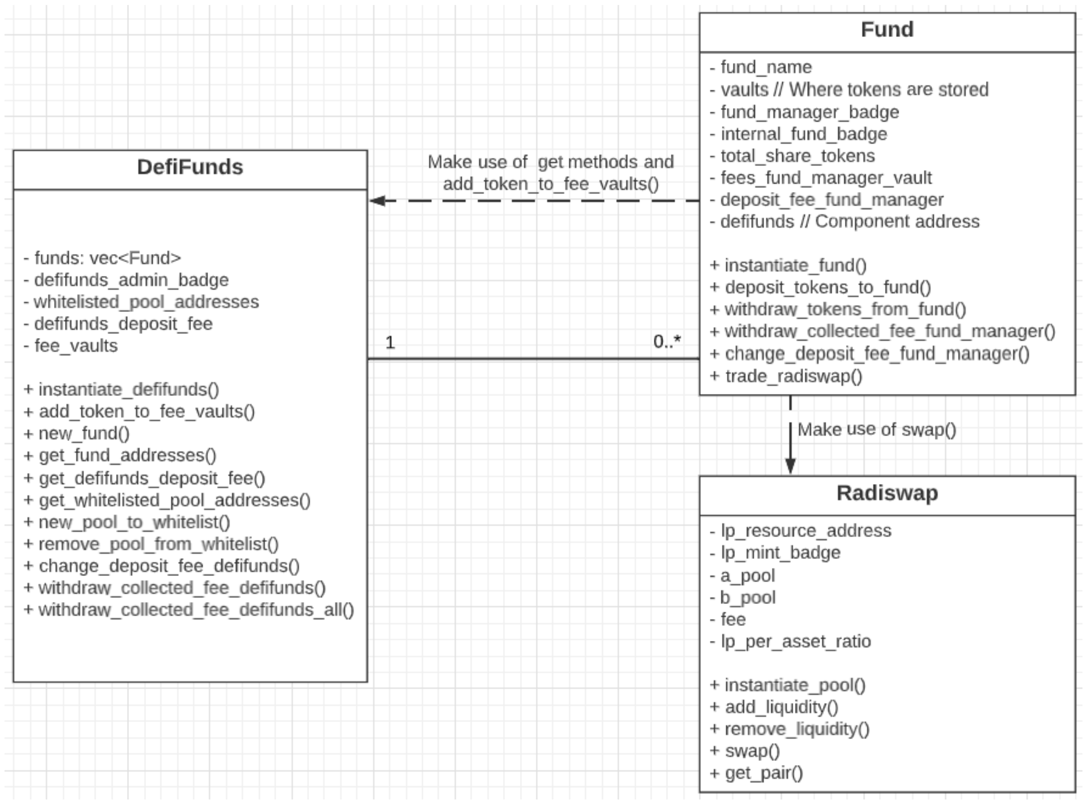
Background:
State of DefiFunds at the start of this Master's thesis

I started this master's thesis with a proof of concept blueprint of DefiFunds. You can see exactly where I ended the specialization project and started my master's thesis in the specialization project branch on my GitHub [22], or in my specialization project [24]. The proof of concept was made with scrypto version 0.6, and I where only working on making the blueprints in my specialization project. The rest of the Dapp was planned to be made at a later point. I will, in this background chapter, give a brief overview of the blueprints from my specialization project.

4.1 Blueprints

The proof of concept Dapp I started with in this master's thesis consists of three blueprints: Fund, DefiFunds, and Radiswap. Fund and DefiFunds were made by me in my specialization project, and Radiswap was made by members of the Radix team [47]. Radiswap is a simple example of an AMM and was valuable to use in a proof of concept, but it will not be used in the final version since an AMM live on the Radix network will need to be used. In this section, I will explain the most important parts of Fund and Defifunds, but I will not go into detail on how Radiswap works as it was not made by me. You can find the code files for where I ended my specialization project on my GitHub in a specific branch named specializationProject [22]. This branch has the code files for scrypto version 0.6.

Figure 4.1 shows how the blueprints work together. The fund blueprint makes use of some methods from DefiFunds and depends on "whitelisted_pool_addresses" and "defifunds_deposit_fee". Fund also makes use of the swap method from the Radiswap blueprint. The DefiFunds blueprint holds a list of all the funds created. I will, in the following two sections, explain the functions and methods that were made. I will also include a reference to where you can find the function/method in the appendix. Note that the functions/methods in the appendix are updated to scrypto version 0.8.



DefiFunds
- funds: vec<Fund>
- defifunds_admin_badge
- whitelisted_pool_addresses
- defifunds_deposit_fee
- fee_vaults
+ instantiate_defifunds()
+ add_token_to_fee_vaults()
+ new_fund()
+ get_fund_addresses()
+ get_defifunds_deposit_fee()
+ get_whitelisted_pool_addresses()
+ new_pool_to_whitelist()
+ remove_pool_from_whitelist()
+ change_deposit_fee_defifunds()
+ withdraw_collected_fee_defifunds()
+ withdraw_collected_fee_defifunds_all()
Make use of get methods and add_token_to_fee_vaults()
1
0..*
Fund
- fund_name
- vaults // Where tokens are stored
- fund_manager_badge
- internal_fund_badge
- total_share_tokens
- fees_fund_manager_vault
- deposit_fee_fund_manager
- defifunds // Component address
+ instantiate_fund()
+ deposit_tokens_to_fund()
+ withdraw_tokens_from_fund()
+ withdraw_collected_fee_fund_manager()
+ change_deposit_fee_fund_manager()
+ trade_radiswap()
Make use of swap()
Radiswap
- lp_resource_address
- lp_mint_badge
- a_pool
- b_pool
- fee
- lp_per_asset_ratio
+ instantiate_pool()
+ add_liquidity()
+ remove_liquidity()
+ swap()
+ get_pair()

Figure 4.1: UML digram showing the relation between the blueprints.

4.1.1 Fund

The main purpose of this blueprint is to create a single fund where normal users can deposit and withdraw tokens. The fund manager can trade with the assets in the fund, but not be able to withdraw the assets to his own account. The Fund blueprint also uses a whitelist from the DefiFunds blueprint that decides what trading pools the fund manager can trade with. This is to make sure that the fund manager doesn't trade with malicious trading pools that could result in impermanent loss for the users. The Fund blueprint also gives the fund manager a fee based on the value deposited to the fund. Down below is a list of the main methods in the Fund blueprint:

deposit_tokens_to_fund Allows normal users to deposit tokens to the fund. When you use this method you need to deposit tokens in almost the same ratio as the fund. If there exist 20 BTC and 10 ETH in the fund, you can for example deposit 2 BTC and 1 ETH. From a user perspective, it would make more sense to send one token type instead of many different ones, but this functionality was not implemented at this stage.
Line 149-186, A.1

witdraw_token_from_fund Allows normal users to send share tokens back to the fund component and receive tokens equal to the share they deposited. They will receive a share of all the different tokens in the fund. From a user perspective, it would make more sense to receive one token type instead of many different ones, but this functionality was not implemented at this stage.
Line 191-210, A.1

withdraw_collected_fee_fund_manager Allows the fund manager to withdraw share tokens from the vault, which contains the collected fee.
Line 267-269, A.1

change_deposit_fee_fund_manager Allows the fund manager to change the fee that goes to the fund manager vault when a user deposits tokens to the fund.
Line 271-274, A.1

trade_radiswap Allows the fund manager to trade with all tokens in the fund. This method made use of the swap method from the Radiswap blueprint. The fund manager can only trade using pools defined in the whitelist. This method is replaced by a new method named "trade_beakerfi".
Line 278-300, A.1

4.1.2 DefiFunds

The main purpose of the DefiFunds blueprint is to keep track of the different funds and control the whitelist that the different funds make use of. There is introduced a time delay of 7 days on all new pools added to the whitelist. This gives users time to withdraw their tokens in case the admin acts dishonestly. If the admin account gets compromised and malicious pools are added, it will give the admin time to call the "remove_pool_from_whitelist" method and warn users to withdraw their tokens. The DefiFunds blueprint also collects a fee to the owner of DefiFunds. Here is a list of the main functions and methods in the DefiFunds blueprint:

instantiate_defifunds This function creates the DefiFunds component. The caller of this function will receive a badge that is used to control the functions that only the DefiFunds admin has access to.
Line 22-69, A.2

new_fund Allows people to create a new fund. The creator of this fund will receive a fund manager badge that is used to control the methods that only the fund manager has access to. It makes use of the "instantiate_fund" function in A.1 line 44-117.
Line 97-120, A.2

new_pool_to_whitelist Allows the DefiFunds admin to add a trading pool to the whitelist. The trading pool will only be valid after 300 epochs (about seven days).
Line 144-146, A.2

remove_pool_from_whitelist Allows the DefiFunds admin to remove a trading pool from the whitelist.
Line 148-150, A.2

change_deposit_fee_defifunds Allows the DefiFunds admin to change the fee that goes to the DefiFunds admin vault when a user deposits tokens to a fund.
Line 152-155, A.2

withdraw_collected_fee_defifunds_all Allows the DefiFunds admin to withdraw the collected fee he has gotten from users depositing to funds.
Line 160-166, A.2

Results

5.1 Requiremenst

This requirements specification provides a brief summary of additional functionality required to develop a basic and functional Dapp. chapter 1 provides an overview of the Dapp's general concept and my underlying motivation. The chosen requirements are based on essential features absent from my previous specialization project. Here is a list of these requirements:

- A user should be able to buy shares in a fund using a single token type.
- A user should be able to sell shares in a fund and receive a single token type in return.
- A user should be able to interact with all the methods in the blueprint via the frontend.
- A user should be able to see info about all the funds in DefiFunds as well as info about his portfolio on the frontend.

5.2 Blueprints - Design and Implementation

DefiFunds consists of two blueprints: DefiFunds and Fund. It also imports two methods from BeakerFi, which is an AMM on Radix. The two methods come from two different blueprints made by BeakerFi. The difference between them is that one method swaps on a pool level, while the other swaps on a DEX level. This chapter will explain the most important parts of the new and replaced methods. The functions and methods that were already made in my specialization project are explained in chapter 4.

Figure 5.1 shows the relation between the blueprints as well as what state variables it holds marked by a (-), and what functions or methods it includes marked by a (+).

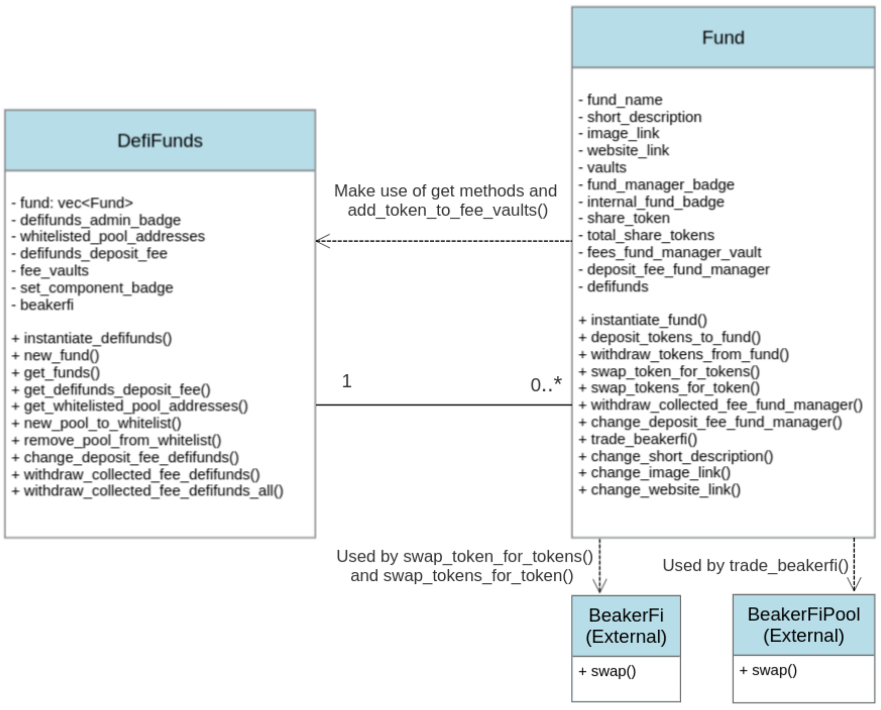

Fund
- fund_name
- short_description
- image_link
- website_link
- vaults
- fund_manager_badge
- internal_fund_badge
- share_token
- total_share_tokens
- fees_fund_manager_vault
- deposit_fee_fund_manager
- defifunds
+ instantiate_fund()
+ deposit_tokens_to_fund()
+ withdraw_tokens_from_fund()
+ swap_token_for_tokens()
+ swap_tokens_for_token()
+ withdraw_collected_fee_fund_manager()
+ change_deposit_fee_fund_manager()
+ trade_beakerfi()
+ change_short_description()
+ change_image_link()
+ change_website_link()
DefiFunds
- fund: vec<Fund>
- defifunds_admin_badge
- whitelisted_pool_addresses
- defifunds_deposit_fee
- fee_vaults
- set_component_badge
- beakerfi
+ instantiate_defifunds()
+ new_fund()
+ get_funds()
+ get_defifunds_deposit_fee()
+ get_whitelisted_pool_addresses()
+ new_pool_to_whitelist()
+ remove_pool_from_whitelist()
+ change_deposit_fee_defifunds()
+ withdraw_collected_fee_defifunds()
+ withdraw_collected_fee_defifunds_all()
Make use of get methods and add_token_to_fee_vaults()
1
0..*
Used by swap_token_for_tokens() and swap_tokens_for_token()
Used by trade_beakerfi()
BeakerFi (External)
+ swap()
BeakerFiPool (External)
+ swap()

Figure 5.1: UML diagram showing the relation between the blueprints

5.2.1 Fund

The Fund blueprint has got some new methods and replaced an old trading method that used the Radiswap blueprint. The old functions and methods will look slightly different from the specialization project since the blueprint is updated from Scrypto version 0.6 to 0.8. You can see the full implementation of the blueprints in Section A.1.

swap_token_for_tokens This method is callable by everyone, and allows people to swap from one token type into many token types at a specified ratio. It is mainly used in combination with "deposit_tokens_to_fund" since you need all the different token types in the fund before you call the "deposit_tokens_to_fund" method. It uses the swap method on the AMM level since there is no need to limit the swapping to specific pools.
Line 213-241, A.1

swap_tokens_for_token This method is callable by everyone, and allows people to swap many token types into one token type. It is mainly used in combination with "withdraw_tokens_from_fund" since you get all the token types in the fund when you call the "withdraw_tokens_from_fund" method. It uses the swap method on the AMM level since there is no need to limit the swapping to specific pools.
Line 245-259, A.1

trade_beakerfi "trade_radiswap" is replaced by this method. It now uses an external swap method made by BeakerFi to facilitate a swap on their AMM. It uses the swap method on the pool level, because the whitelist in the DefiFunds blueprints limits the trading on pool addresses.
Line 278-300, A.1

change_short_description Allows the fund manager to change the description of the fund. Line 302-304, A.1

change_image_link Allows the fund manager to change the image associated with the fund. Line 306-308, A.1

change_website_link Allows the fund manager to change the website associated with the fund. Line 310-312, A.1

5.2.2 DefiFunds

DefiFunds is mostly the same blueprint as it was in my specialization project. It is updated from the Scrypto version 0.6 to 0.8, so it will look slightly different from the code file in my specialization project. The only real change worth mentioning is a new method named "set_address," which is only accessible by the component itself. It is used for a workaround updating from 0.6 to 0.7. This is discussed more about in Section 6.1.1. You can see the full implementation of the blueprint in Section A.1.

5.3 Frontend - Design and Implementation

I will in this section explain what I did related to the frontend. I built a full stack solution similar to the full stack solution made by RDX Works [50]. The way that the wallet connects to the Dapp and sends manifest stubs to the wallet is heavily inspired by this example. I have made manifest stubs for all functionality needed on the frontend. I have also made a way to get close to all the data needed for the frontend.

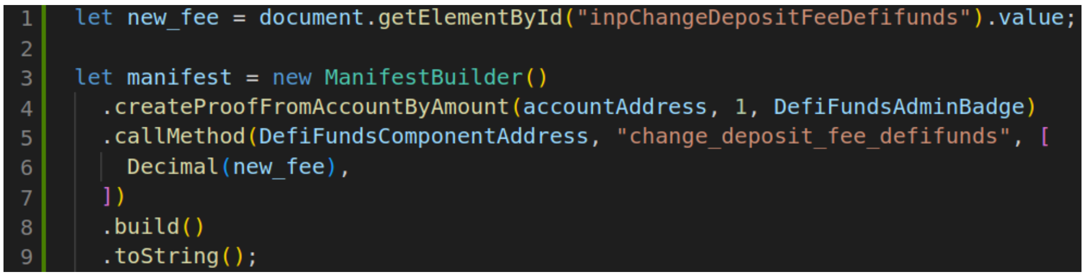
My expertise is not in frontend development, so my focus hasn't been on creating a user-friendly frontend. I am unfamiliar with React or similar frontend languages, so my priority has been making the Javascript functionality related to the Radix Network. My plan is to team up with a frontend developer to complete the Dapp. In the coming subsection, I will go through the Javascript files related to the manifest, the API calls, and the general functions to get the data needed for the frontend; see the UI design Section 5.4.

5.3.1 Manifest stubs

An essential part of the frontend-related work was to create manifest stubs. Those are made so that a user can input parameters, send the manifest stub to his wallet, and finally sign and send the transaction to the network. In this subsection, I will go through three of the manifests; change fee, deposit, and withdraw. All the manifests are in the "index.js" file; see Section B.2.

Change fee

This manifest stub lets the creator of DefiFunds change part of the total deposit fee associated with users depositing to a fund. The manifest stub makes use of "change_deposit_fee_defifunds" from the DefiFunds blueprint.



```
1  let new_fee = document.getElementById("inpChangeDepositFeeDefifunds").value;
2
3  let manifest = new ManifestBuilder()
4    .createProofFromAccountByAmount(accountAddress, 1, DefiFundsAdminBadge)
5    .callMethod(DefiFundsComponentAddress, "change_deposit_fee_defifunds", [
6      Decimal(new_fee),
7    ])
8    .build()
9    .toString();
```

Figure 5.2: Manifest stub for changing a fee

Line 1: The user input is taken from a field value on the frontend.

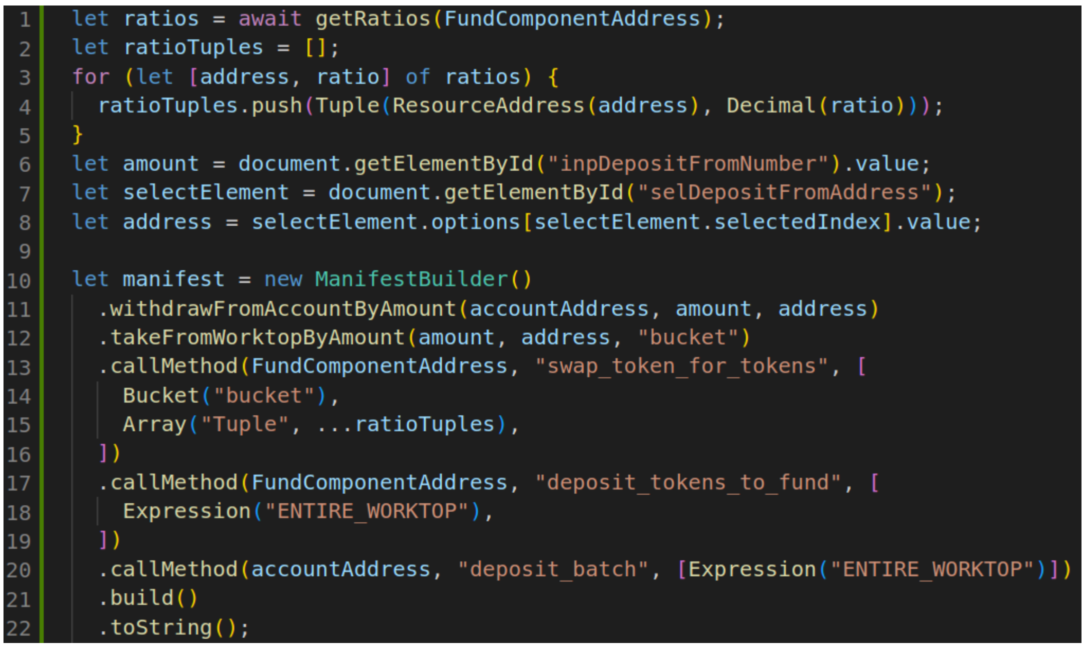
Line 4: A proof is created to show that the user is the DefiFunds Admin. "AccountAddress" is the address of the user that is logged into the Dapp, and the DefiFundsAdminBadge is a constant address given when DefiFunds was instantiated.

Lines 5-7: The change fee method is called on the DefiFundsComponent. The DefiFundsComponetAddress is the address that was created when DefiFunds was instantiated.

Lines 8-9: The manifest stub is made ready before it can be sent to the wallet.

Deposit to fund

The deposit manifest stub lets a user buy shares of a fund. The manifest stub uses two methods from the fund blueprint; "swap_tokens_for_token" and "deposit_tokens_to_fund."



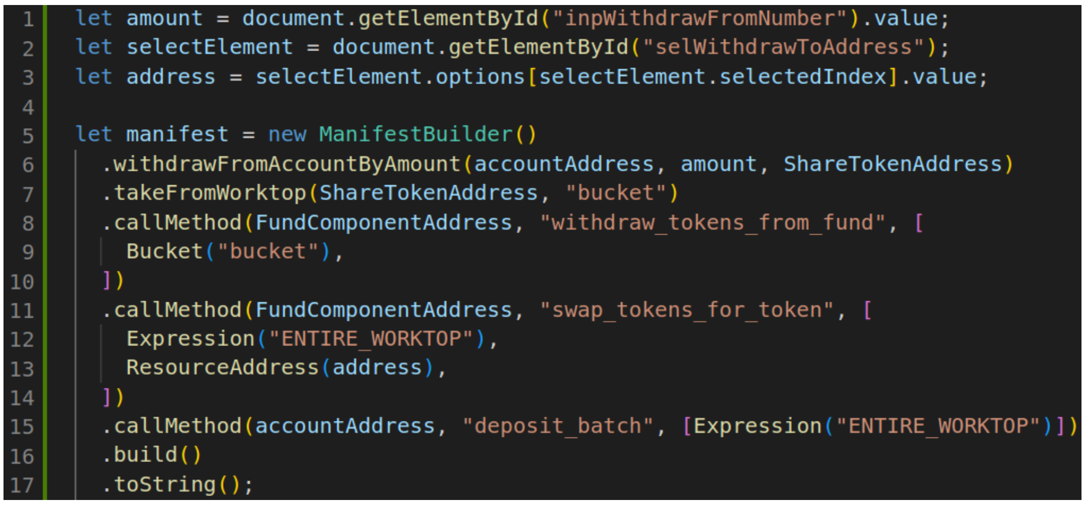
```
 1  let ratios = await getRatios(FundComponentAddress);
 2  let ratioTuples = [];
 3  for (let [address, ratio] of ratios) {
 4    ratioTuples.push(Tuple(ResourceAddress(address), Decimal(ratio)));
 5  }
 6  let amount = document.getElementById("inpDepositFromNumber").value;
 7  let selectElement = document.getElementById("selDepositFromAddress");
 8  let address = selectElement.options[selectElement.selectedIndex].value;
 9
10  let manifest = new ManifestBuilder()
11    .withdrawFromAccountByAmount(accountAddress, amount, address)
12    .takeFromWorktopByAmount(amount, address, "bucket")
13    .callMethod(FundComponentAddress, "swap_token_for_tokens", [
14      Bucket("bucket"),
15      Array("Tuple", ...ratioTuples),
16    ])
17    .callMethod(FundComponentAddress, "deposit_tokens_to_fund", [
18      Expression("ENTIRE_WORKTOP"),
19    ])
20    .callMethod(accountAddress, "deposit_batch", [Expression("ENTIRE_WORKTOP")])
21    .build()
22    .toString();
```

Figure 5.3: Manifest stub for depositing to a fund

Line 1:	This calculation determines the amount of the deposit token needed to purchase each token type, in order to maintain the same output ratio as the fund after performing swaps. Explained in more detail in Section 5.3.2.
Line 3-5:	A specific syntax on the ratios is required by the manifestBuilder.
Line 6-8:	The user inputs are taken from the field values on the frontend.
Line 11-12:	The amount that the user wants to deposit is taken from his account to the worktop, and from the worktop into a bucket.
Line 13-16:	The "swap_token_for_tokens" method is called, and the one token type is swapped into the token types needed to deposit to the fund. The tokens are placed on the worktop after the swaps.
Line 17-19:	The "depsoit_tokens_to_fund" method is called with all tokens present on the worktop as a parameter. Share tokens representing the deposited amount are returned to the worktop.
Line 20:	The share tokens on the worktop are deposited into the user account.
Line 21-22:	The manifest stub is made ready before it can be sent to the wallet.

Withdraw from fund

The withdraw manifest stub lets a user sell shares of a fund. The manifest stub uses two methods from the Fund blueprint; "withdraw_tokens_from_fund" and "swap_tokens_for_token." Something to note with this manifest stub is that it didn't work in 0.8, likely because of a bug on Radix's side with "Expression("ENTIRE_WORKTOP")," which is discussed in Section 6.1.4.



```
 1  let amount = document.getElementById("inpWithdrawFromNumber").value;
 2  let selectElement = document.getElementById("selWithdrawToAddress");
 3  let address = selectElement.options[selectElement.selectedIndex].value;
 4
 5  let manifest = new ManifestBuilder()
 6    .withdrawFromAccountByAmount(accountAddress, amount, ShareTokenAddress)
 7    .takeFromWorktop(ShareTokenAddress, "bucket")
 8    .callMethod(FundComponentAddress, "withdraw_tokens_from_fund", [
 9      Bucket("bucket"),
10    ])
11    .callMethod(FundComponentAddress, "swap_tokens_for_token", [
12      Expression("ENTIRE_WORKTOP"),
13      ResourceAddress(address),
14    ])
15    .callMethod(accountAddress, "deposit_batch", [Expression("ENTIRE_WORKTOP")])
16    .build()
17    .toString();
```

Figure 5.4: Manifest stub for withdrawing from a fund

Line 1-3:	The user inputs are taken from the field values on the frontend.
Line 6-7:	The amount of share tokens that the user wants to sell is taken from his account to the worktop, and from the worktop into a bucket.
Line 8-10:	The "withdraw_tokens_from_fund" method is called with the share tokens as a parameter. Tokens from the fund according to that share are returned to the worktop.
Line 11-14:	All tokens present on the worktop are swapped into a specified token type.
Line 15:	The tokens gotten from the swaps are deposited to the user account.
Line 16-17:	The manifest stub is made ready before it can be sent to the wallet.

5.3.2 "getRatios" - a function needed for "swap_token_for_tokens"

"getRatios" is a function that should calculate the ratio used as inputs for "swap_token_for_tokens" to receive tokens after the swap equal to the ratio in the fund. You can see the implementation of the function in Section B.3 line 329-348. It makes use of API calls to BeakerFi to get the price of the different tokens in the DEX. It also makes an API call to the Radix network to get the token amounts in the fund you are requesting for. Why I decided to get the price from a centralized source and request the tokens amounts on the frontend and not in the smart contract can be read about in Section 6.4.

The "getRatios" function I have made for now assumes that there isn't any slippage when doing a swap. This is good enough on smaller swaps, but on a swap with a relatively high value compared to the swapping pool there would be significant slippage. I first thought that BeakerFi was basic and used a simple automated market maker with the constant product function $x * y = k$, as explained in Section 2.3 under "Decentralized exchanges." I explored with a maximizing function to get "getRatio" as approximately as possible. I figured out the theory, but met on some problems when trying to implement the function in Javascript. I also realized there was no point in making this yet, as BeakerFi didn't use that constant product formula, and I didn't know what kind of formula the swapping Dapp that I was going to use at the end was going to use. I decided to leave the function as it is for now and fix it at a later point.

5.3.3 Data needed for the frontend

Another essential part of the frontend is the API calls that let users get the info needed for a user-friendly frontend. As I am not a frontend developer, I wasn't fully aware of the best practices to limit API calls to what's necessary. I decided that handling the caching or storing of API calls would be better suited for a frontend developer, so my priority has been to identify and locate the required data.

The main file for API calls is the javascript file named apiDataFetcher; see Section B.3. This file is a general file that does all the API calls, stores the info in some global variables, and has lots of get functions to get the specific data needed from those global variables. The get functions take information both directly from the global variables and do calculations on that data. The functions in this file are either fetch, update or get functions.

Fetch functions

The fetch functions are direct API calls; see B.3 line 51-140. They do state calls to the Radix network; on Defifunds to retrieve info about what funds exist, on Funds to retrieve info about the fund, and on account addresses to retrieve information about tokens in the user wallet. There are also API calls to BeakerFi to retrieve the amount of tokens in a pool, which is later used for calculating a token's price. An API call to Coingecko, a cryptocurrency data aggregator, is also done to retrieve the price of XRD since you only could calculate the price relative to XRD and not USD with BeakerFi's API.

Update functions

The update functions, see B.3 line 142-189, make use of the fetch functions to update the global variables. This is done to prevent doing API calls every time you need the data. You can call these functions as often as you need updated data. There are four global variables:

funds:	An array with all the fund addresses in DefiFunds.
fundsInfo:	A map with fund addresses and the corresponding data for the fund, like fund name, what tokens it contains, etc.
tokensInWallet:	A map with all the token addresses and the corresponding amount to the wallet that is logged into the Dapp.
tokenPrices:	A map with all the token addresses on BeakerFi and the corresponding price in USD.

Get functions

The get functions, see B.3 line 191-348, are used to get the specific data needed for the frontend from the global variables. This can simply be taking a value from a map, or it can require some calculations on multiple values to get the specific data needed.

Javascript files specific for a UI page

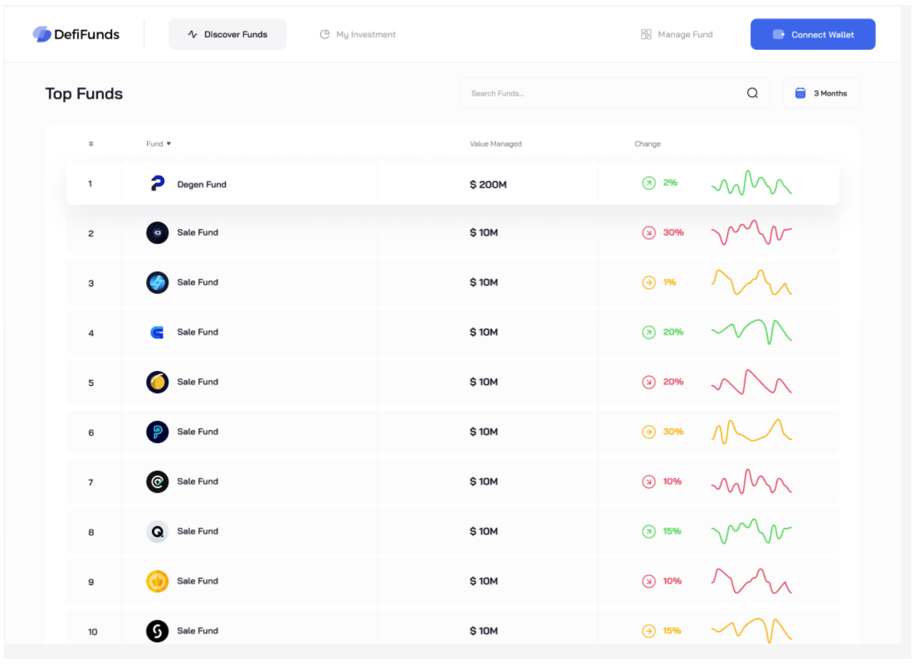
Apart from the get functions in the "apiDataFetcher" file, there are also four other files that have get functions. The get functions in "apiDataFetcher" are more general functions, while those in the four other files are more specific for the different frontend sites. The code implementation of those files and the corresponding UI design can be found here:

Discover Funds:	B.4 - Figure 5.5
Fund:	B.5 - Figure 5.6
My Investments:	B.6 - Figure 5.7
Manage Fund:	B.7 - Figure 5.8

Most of the functionality is made, but there still lack some features. The historic price data for the funds is not fully implemented yet. I am thinking of storing this price data in a database myself. You can take a look at my GitHub for where I am in the process [25]; go back one folder and move into historyDB. Historic portfolio data is also not implemented yet.

5.4 UI design

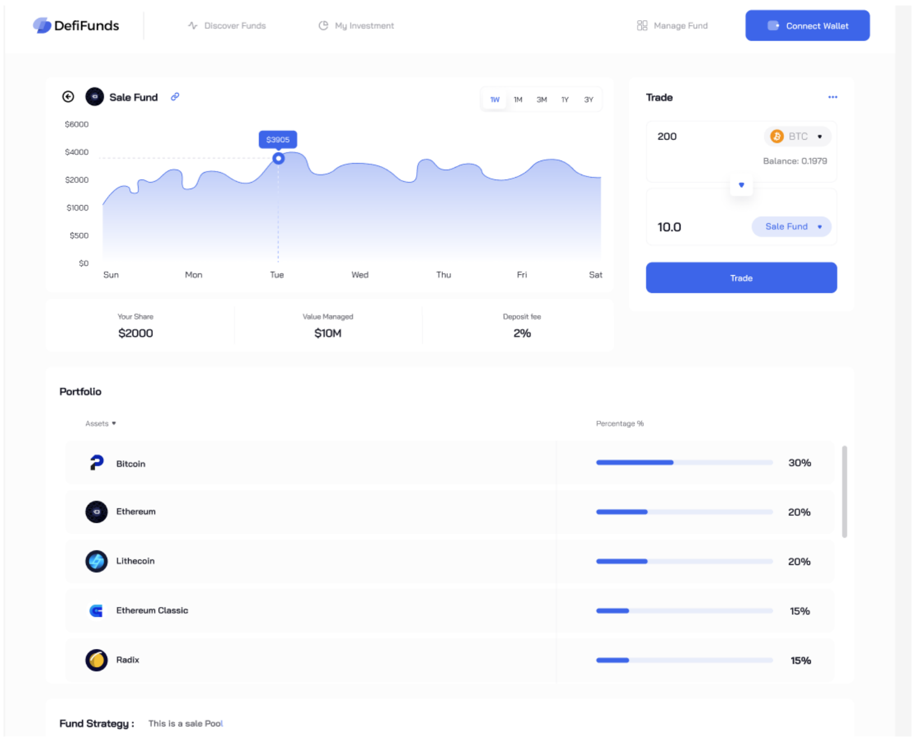
The UI design is made by a freelancer I hired. My part of the work related to the UI design has been to draw sketches of the design and give specifications of what needed to be on the website to make it compatible with the frontend integrations I was thinking of making and already had made. My work was also to give ideas and feedback while he was doing the design. We had some voice calls, but the feedback on the design mainly consisted of comments using Figma. There I could comment directly on the design, which made the communication process a lot easier. You can go into the Figma file, create an account, go to the message settings, and select "show resolved comments" to see the comment log. I also adjusted simpler stuff myself, for example, how the text was written. The five design pages are shown below and in the Figma file [15].



Top Funds

#	Fund	Value Managed	Change
1	Degen Fund	$ 200M	2%
2	Sale Fund	$ 10M	30%
3	Sale Fund	$ 10M	1%
4	Sale Fund	$ 10M	20%
5	Sale Fund	$ 10M	20%
6	Sale Fund	$ 10M	30%
7	Sale Fund	$ 10M	10%
8	Sale Fund	$ 10M	15%
9	Sale Fund	$ 10M	10%
10	Sale Fund	$ 10M	15%

Figure 5.5: Discover Funds
Showing all the funds you can invest in.



200 BTC
Balance: 0.1979
10.0 Sale Fund

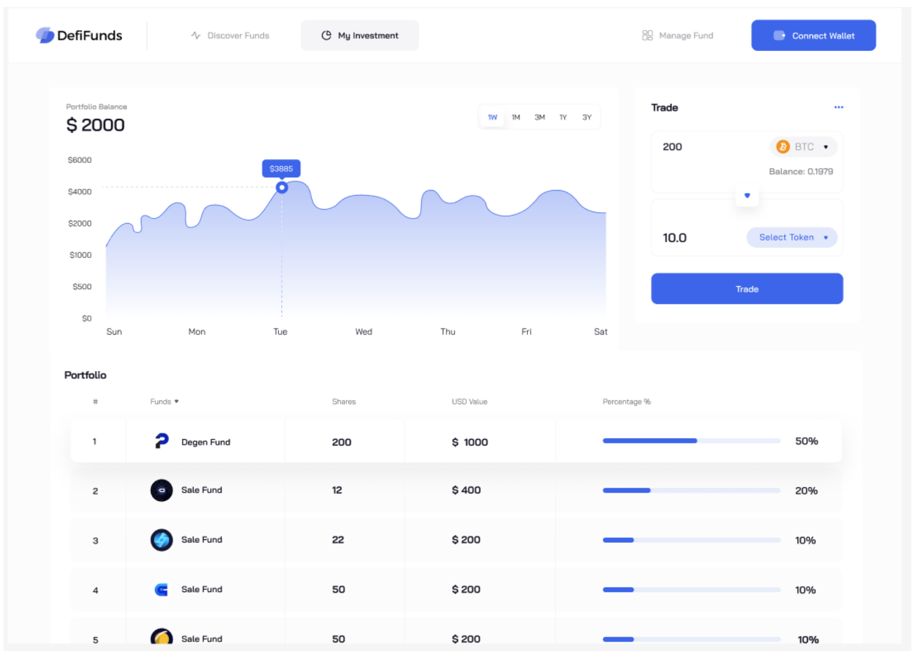
Your Share	Value Managed	Deposit fee
$2000	$10M	2%

Portfolio

Assets	Percentage %
Bitcoin	30%
Ethereum	20%
Lithecoin	20%
Ethereum Classic	15%
Radix	15%

Fund Strategy : This is a sale Pool

Figure 5.6: Fund
Showing a specific fund. You can also buy shares in that fund.



Portfolio Balance
$ 2000

Trade

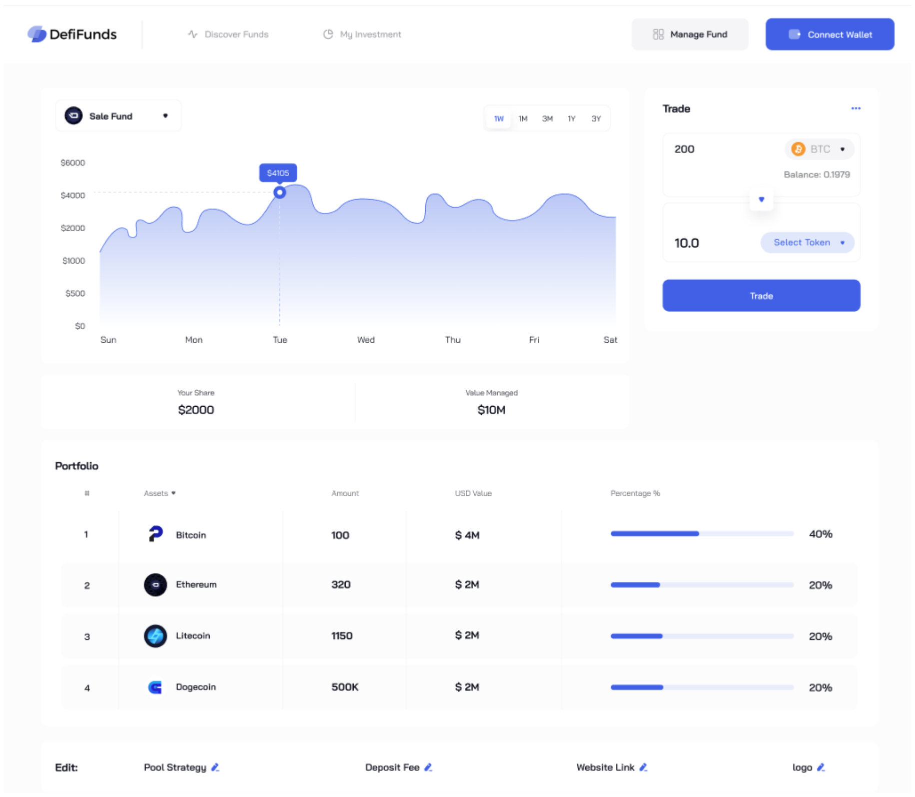
200 BTC
Balance: 0.1979

10.0

Portfolio

#	Funds	Shares	USD Value	Percentage %
1	Degen Fund	200	$ 1000	50%
2	Sale Fund	12	$ 400	20%
3	Sale Fund	22	$ 200	10%
4	Sale Fund	50	$ 200	10%
5	Sale Fund	50	$ 200	10%

Figure 5.7: My Investments
An overview of all the funds you have invested in. You can also buy shares in different funds.



Trade

200 BTC
Balance: 0.1979

10.0

Your Share
$2000

Value Managed
$10M

Portfolio

#	Assets	Amount	USD Value	Percentage %
1	Bitcoin	100	$ 4M	40%
2	Ethereum	320	$ 2M	20%
3	Litecoin	1150	$ 2M	20%
4	Dogecoin	500K	$ 2M	20%

Edit: Pool Strategy Deposit Fee Website Link logo

Figure 5.8: Manage Fund
A place where the fund manager can see the status of his fund, adjust info on it and trade with the tokens in the fund.

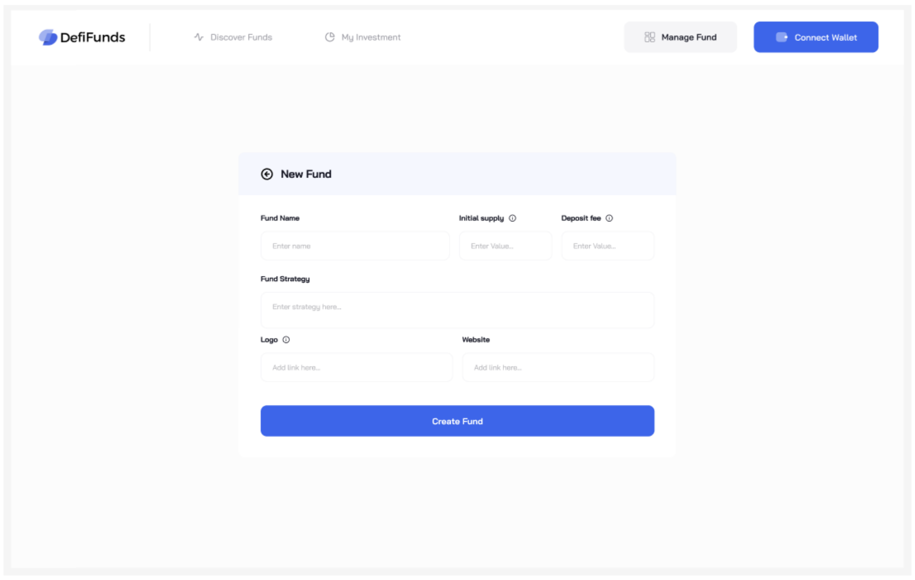


Figure 5.9: New Fund
A way to create a fund for a new fund manager. This page will appear when the Manage Fund button is pressed if the user doesn't already manage a Fund.

5.5 Triumph in the Scrypto DeFi Challenge

The Scrypto DeFi Challenge was a significant milestone in the development of my Dapp. The challenge aimed to inspire developers and researchers to build the next billion-dollar Dapp using Scrypto on the Radix platform. [16] This section will recount my experience participating in the challenge and winning an honorable mention prize worth $3,500. You can find my contribution here [23].

5.5.1 Overview of the challenge

The Scrypto DeFi Challenge was organized by Devpost and managed by RDX Works. It featured $50,000 in prizes, awarded in the form of XRD tokens. The judges evaluated the submissions based on the quality of the code, asset-oriented design patterns, functionality, creativity, and documentation.

The challenge was to build a Scrypto blueprint with DeFi functionality and frontend code to let users interact with the Dapp on the Betanet. You could enter the competition both as an individual or as a team. Some bonus categories required you to build specific types of Dapps, but the main challenge had an open category.

5.5.2 Experience participating in the challenge

Having a smaller and more concrete goal than to build the Dapp to production helped a lot. It felt good to have a deadline and that you knew the current version of Scrypto and betanet wouldn't change during the competition period. I was focused solely on developing in that period and didn't prioritize to start writing on my master's thesis before the competition had ended. I had never entered in a hackathon before, and I was pretty excited about it as I felt I had a real chance of delivering a good submission.

During the competition, I probably found a bug with Scrypto or the betanet itself when trying to send a transaction. The logic seemed right in the transaction manifest, and I could not understand what I was doing wrong. I even hired talesOfBeam, a known expert in the Radix Community, to try to find out what I did wrong. After lots of testing, we found out that it probably was a problem with Scrypto or the betanet and talked with Jake from RDX Works so that they could fix it before the next release of Scrypto. This is more closely discussed in Section 6.1.4.

5.5.3 Testing the application

The Dapp I made was live on betanet, but betanet is no longer up and running. As of writing, I have not updated the Dapp to the current version of the test network, but that might have changed when you read this. You can check my GitHub for the most up-to-date version [25]. The main reason for not having it up to date as of writing is that my Dapp depends on BeakerFi, which is not up and running. If you want to test some basic functionality in a simulator, you can check out the demo section in my specialization project [24].

Discussion

I will in this chapter have a discussion on the challenges I have had during my work related to the master's thesis. If you would like to read more discussion related to the security of the blueprint I would encourage you to read two sections from my specialization project; "4.3.3 Examples - Misusing methods" and "5.2 Weakness with the whitelist" [24].

6.1 Challenges associated with developing on an early stage network

I will in this section go through some challenges I have had regarding developing on an early-stage network. There have been some breaking changes going from Scrypto version 0.6 to 0.8. There have also been some challenges regarding a limited API, some bugs on Radix's side, and poor debugging capabilities.

6.1.1 Scrypto version 0.6 - 0.7

It wasn't possible to get the component address of it selves anymore in 0.7. I used it in 0.6 to send the component address of DefiFunds over to the Fund blueprint when creating a new fund; see Section A.2 line 111. The Fund blueprint needed to have the component address of DefiFunds to be able to know where the fees related to the DefiFunds admin should be sent to, and to know what pools are in the whitelist. In 0.6 you could simply call "Runtime::actor().as_component().0", but in 0.7 this wasn't possible anymore. I discussed it with several others in the Scrypto community, and none of us found an easy alternative. We ended up with a workaround for 0.7 where we saved the component address of itself in the state of the component; see Section A.2 line 30-36, 65, and 71-77. It was pretty cumbersome and looked more complex than necessary, but none of the ones I discussed with could find other ways to achieve the same. I guess this is one of the downsides of developing on a language that is under constant development and has breaking changes.

6.1.2 Scrypto version 0.7 - 0.8

Updating the Scrypto blueprint from 0.7 to 0.8 was relatively straightforward, but the testing part was worse. There were some bugs in the simulator with how you switched accounts. In 0.7, you created a new account by calling "resim new-account," and received a public and private key in return. To switch account you used "resim set-default-account" and put the private and the public key as parameters. In 0.8, a badge was also included in creating a new account since you needed it to publish the package with the blueprint. It was a badge used to collect royalties for blueprint creation. For switching an account in resim you needed all three parameters, public key, private key, and the badge. The problem occurred with a bug in 0.8 where you only got a badge from the first account you created and not the other ones. Because of this bug, there wasn't a straightforward way to create multiple accounts and switch between them. Someone in the community found workarounds for it, but I just dropped updating the testing in the resim from here, since I knew it worked in 0.7 and it was relatively straightforward to update. I thought that it wouldn't be a need for excessive testing on the blueprint level anymore, at least for now.

I knew I also couldn't test it fully in the simulator anyway since I only had the method signatures and not the code for the swapping methods from BeakerFi; see Section A.1 line 4-21. I could only test my methods, which didn't include the external import components, locally. The methods that included the external import components, needed to be tested on the betanet. I deployed the Dapp to betanet and tested it there. It took longer time to test it on the betanet, than using resim to test it, but I was pretty sure that I didn't need to test the methods that much. I could have decided to create a simple swapping blueprint with the same method signatures and used that to test it locally, but I estimated that it would in total take more time, and ended up not creating it.

6.1.3 Lack of gateway API funcionality

One particular functionality was noticeably absent. There was no way to retrieve the information regarding the amount of a resource contained within a vault. The system allowed the retrieval of the resource amount at the component level, but not at the vault level. This limitation led me to modify the blueprint to incorporate an additional "Decimal" field value on the vault; see Section A.1 line 32. Subsequently, I had to update this decimal value every time a transaction affected the vault, i.e., when resources were deposited into or withdrawn from the vault.

This modification will be reverted when the necessary API for this functionality becomes available. The reason for not waiting until the API functionality became available, was primarily driven by my intent to deliver a good product for the hackathon.

6.1.4 Bugs on Radix's side and limited debugging possibilities

One particular bug, likely on Radix's side, took a lot of time to figure out. There was probably an edge case bug with "Expression("ENTIRE_WORKTOP")" that made my withdraw manifest to fail; see Section B.2 line 334-346. The expression is supposed to turn all resources on the worktop into a vector with buckets. After discussing with a community member called talesOfBeam and Jake from RDX Works we concluded that the problem likely was that "Expression("ENTIRE_WORKTOP")" included empty buckets in the vector. My "swap_tokens_for_token" method, see Section A.1 line 245-259, takes in a vector with buckets, but the buckets can only consist of specific resource addresses to not fail. More precisely, it can only handle resources that are available to swap on BeakerFi. The problem was likely that the expression included an empty bucket with the share token. This will cause my method to fail since the share token is not one of the resources that can be swapped on BeakerFi. This made the withdraw manifest not to work on Betanet. Jake from RDX Works thinks he has solved the bug for RCnet, but it has not been tested yet, as far as I know. Neither me nor BeakerFi have updated their blueprints and published it to RCnet as of writing.

Another problem that did it harder to debug was the poor error messages from the network and that there wasn't any documentation for the error message. An example of the error message I got when I tried to send the deposit manifest was: "KernelError(WasmRuntimeError(InterpreterError("Trap(Trapkind:Unreachable)")))".

6.2 Time priorizization

Prioritizing what to build, what to outsource, and when to build stuff has also been quite challenging. Some of the challenge comes from developing on an early-stage network. Another part of the challenge comes from my uncertainty in my skill level, and what is easy to learn and what is harder to learn.

It was often a question if the problem that I tried to fix will fix itself in the next update of Scrypto. And even if I knew it was going to be fixed, is it worth making a workaround for this version, or not? An example is the bug with the radix engine simulator discussed in Section 6.1.2. I knew that this bug likely was going to be fixed in the next version, but I wasn't sure if I should use my time on fixing it or just wait for the next update. Another one is the changed functionality in Section 6.1.1. Was this functionality changed to output something else because they wanted you to change design patterns, or was it changed to something else and they just forgot to think about which design patterns it had been used for in earlier versions?

It was also hard to decide if I should prioritize time on finding the correct API request, as I knew that it was not going to be stable. Will it be stable enough that the work I do will help a lot, or is it just a waste of time? My goal of completing the Dapp for the scrypto challenge helped a lot on this question since I knew that the API would not change before the competition had ended.

Another challenge I had was that I am not that good at frontend development. I knew basic HTML, CSS, and Javascript, but I knew little about React or other similar languages. I figured out that Javascript was what I was most familiar with, and avoided thinking about how the website looked. I was only going to focus on the part that relates to communication with the Radix network and leave the rest of the work to a frontend developer. Since I was not familiar with the best practices of storing the data request with the API, I decided to go for a method I felt was logical with the knowledge I had. I talked with a frontend developer, that told me that React had some good methods of handling the caching and the updating. In retrospect, I would have consulted even more with a frontend developer about how caching and handling updates of API calls worked to find a better solution. You can see how I structured it in Section B.3.

Another part that was a lot easier to decide was what to do with the UI design. I knew that I was not particularly good at graphic design, and I also knew that the design of the website is really important. Hiring a professional to design it with me explaining how it should be done was an easy choice. Deciding who to hire was harder.

6.3 UI design and the communication process

I have hired freelancers from Fiverr for other projects and have learned some lessons. While their portfolio says a lot about their design skills it says very little about their communication skills. Trying to find both good design skills and communication skills for a cheap prize can be quite tricky. The ratings on Fiverr often gives some info, but the communication is often the tricky part anyway. I have learned that it is important to give very clear instructions on what you want and give regular feedback.

It was the first time I got people to make UI design for me, so I wasn't so sure on how that process would work. I provided sketches with details of what to include, as well as several links to websites similar to the style I wanted, so the designer more clearly could understand what style I preferred. I have some experience with sketching designs from a course at NTNU named "Human–Computer Interaction", which made that process easier.

The designer I hired suggested using Figma which proved to be a valuable tool. It was a good tool for the designer to design, and it was also an effective tool for me to give feedback. I could give comments directly on the design which proved to be valuable for simpler communication. We could mostly communicate using comments there, but it was also crucial to have calls at the start of the process, as well as during the process when more complex questions arose.

6.4 Where to implement functionality?

On my specialization project deciding where to implement functionality was relatively straightforward as I only built blueprints. On my master's, I should go from just a blueprint to a fully working Dapp, and the question of where functionality was best implemented was a harder decision. Should I make this in the blueprint, in the frontend, or at the backend?

The functionality I used the most time on was how do I buy the fund. I had one function "deposit_tokens_to_fund", see Section A.1 line 149-186, that took in all tokens in the fund as an input parameter and returned share tokens. The problem was that I can't require a user to have all the tokens needed for the fund to buy a part of the fund. I needed to make a method that takes one token as a parameter and returns the needed tokens.

In the blueprint, I could get info about how many of each token there is in a fund, but I couldn't get the ratio relative to the price I needed for swapping one token into the tokens needed for the fund. I thought about if I could get this info from an oracle, or from the pool ratio to BeakerFi. There weren't any oracles made on Radix and I also knew that they will not be free to use. If the price I needed is critical for the security, I would have needed to use an oracle on a smart contract level. This was not the case because of how I made the "deposit_tokens_to_fund" method. If the ratio is off, the user will get some rest tokes back, but it is not a security problem, only an inconvenience. I therefore knew that I could safely request the token price on the frontend and not on the smart contract level. I ended up making a method called "swap_token_for_tokens", see Section A.1 line 213-241, that took the ratio requested on the frontend as one of the input parameters.

Conclusion

I have built a lot further on the Dapp, and the Dapp is mostly missing parts related to the frontend. I also participated in a hackathon arranged by RDX Works, where I ended up winning an honorable mention prize worth $3,500. Two of the four requirements specified in Section 5.1 are clearly met. The first requirement, "A user should be able to buy shares in a fund using a single token type," is partly met. The user can do so, but if the user buys for a substantial amount of money, the user may receive a fair portion of dust tokens in return in addition to the share tokens. The fourth requirement, "A user should be able to see info about all the funds in DefiFunds as well as info about his portfolio on the frontend," is also partly met. Most of the information needed for the UI design is available, but some is still missing, and the info is not displayed in a user-friendly way.

My main motivation for building this Dapp has been to create something that can be useful for people. A lot of work has been done in this master's, but there is still a substantial amount of frontend work to get the Dapp ready. I am motivated to complete this Dapp, and I will likely team up with or hire a frontend developer to get it ready. I have included two lists that sums up what has been done in this master's and what needs to be done to complete it.

7.1 What has been done

The work that has been done consists of three main parts; Completing the blueprint, communicating with the UI designer, and making a functional, but not user-friendly frontend. Down below is a more detailed list:

- Changed the "trade_radiswap" method to use a method from an external component from BeakerFi.
- Made a method called "swap_token_for_tokens". Combined with the deposit method, it lets users deposit a single token type to the fund.
- Made a method called "swap_tokens_for_token". Combined with the withdraw method, it lets users withdraw from the fund and receive a single token type.
- Deployed the Dapp to betanet.
- Made a frontend that lets a user call all the methods on the Dapp.
- Made a good-looking UI, as well as logo, and acquired the domain name defifunds.io.
- Made lots of javascript functions to acquire the data needed to make the frontend look like the UI.

7.2 Future Work

The main part of the resulting work is related to the frontend. I have listed the essential tasks that need to be done, and some possible tasks that can be done to improve the Dapp.

Essential tasks

- Update the blueprints to the newest version of Scrypto, as well as other minor changes related to API calls discussed in Section 6.1.3.
- Update "getRatios" to take slippage into account.
- Find out a way to get historic price of the funds as well as historic portfolio data.
- Deploy the Dapp to mainnet when it goes live.
- Make a frontend that matches the UI design.

Possible tasks

- Possible to substitute share tokens with NFTs representing the number of share tokens and the value of the share token at the deposit time. Fees that only will be taken if the fund manager trade with profits can then be implemented. For example, take a withdrawal fee of 10% of the profits made relative to when the user deposited and withdrew his tokens.
- Find out if there exists a safer solution to control the whitelist of pools. For example, let a decentralized autonomous organization control the whitelist.
- Get the Dapp audited.

Bibliography

[1] Ethereum TxHash: 0x01afae47b0c98731b5d20c776e58bd8ce5c2c89ed4bd3f8727fad3ebf32e9481. URL: https://etherscan.io/tx/0x01afae47b0c98731b5d20c776e58bd8ce5c2c89ed4bd3f8727fad3ebf32e9481.

[2] Ethereum TxHash: 0x13bfe9bea4e21ea532988350a8b7eef002686b3b655f19f8760210d1069ab585. URL: https://etherscan.io/tx/0x13bfe9bea4e21ea532988350a8b7eef002686b3b655f19f8760210d1069ab585.

[3] *5.2 Ledger State |IOTA*. URL: https://wiki.iota.org/IOTA-2.0-Research-Specifications/5.2LedgerState (visited on 12th Sept. 2022).

[4] Aave. *aave-protocol/Aave_Protocol_ Whitepaper_ v1_ 0 |Aave*. Jan. 2020. URL: https://github.com/aave/aave-protocol/blob/master/docs/Aave_Protocol_Whitepaper_v1_0.pdf.

[5] Eray Arda Akartuna et al. *NFTs and Financial Crime |Elliptic.co*. 2022. URL: https://www.elliptic.co/resources/nfts-financial-crime.

[6] *Bitcoin Energy Consumption Index - Digiconomist*. URL: https://digiconomist.net/bitcoin-energy-consumption (visited on 1st Oct. 2022).

[7] Lorenz Breidenbach et al. *An In-Depth Look at the Parity Multisig Bug*. July 2017. URL: https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/ (visited on 2nd Oct. 2022).

[8] Vitalik Buterin. *Why sharding is great: demystifying the technical properties*. Apr. 2021. URL: https://vitalik.ca/general/2021/04/07/sharding.html.

[9] Florian Cäsar et al. 'A Parallelized BFT Consensus Protocol for Radix'. In: (2020). URL: https://assets.website-files.com/6053f7fca5bf627283b582c2/608811e3f5d21f235392fee1_Cerberus-Whitepaper-v1.01.pdf.

[10] Chainlink. *What Is the Blockchain Oracle Problem? Why Can't Blockchains Solve It?* Aug. 2020. URL: https://blog.chain.link/what-is-the-blockchain-oracle-problem/ (visited on 5th Oct. 2022).

[11] Emanuel Coen. *Learn about Token Allowances & How to keep your Ethereum Secure*. 2020. URL: https://cryptotesters.com/blog/token-allowances (visited on 3rd Oct. 2022).

[12] *Consensus mechanisms |Ethereum.org*. Sept. 2022. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/ (visited on 20th Sept. 2022).

[13] Brady Dale. *Ethereum 2.0's Composability Concerns, Explained - CoinDesk*. Oct. 2020. URL: https://www.coindesk.com/tech/2020/10/13/will-a-sharded-ethereum-be-flexible-enough-for-decentralized-finance/ (visited on 30th Nov. 2022).

[14] *Decentralized Data Model |ChainlinkDocumentation*. URL: https://docs.chain.link/architecture-overview/architecture-decentralized-model (visited on 15th Oct. 2022).

[15] designershub.io. *Figma File*. URL: https://www.figma.com/file/dxSKJUxvfyQ3e2wKqYOtJT/Tobias_eth?type=design&node-id=0%5C%3A1&t=s4wjOnPNkpoz5B0v-1 (visited on 19th May 2023).

[16] Devpost. *Scrypto DeFi Challenge*. 2023. URL: https://scryptodefi.devpost.com/ (visited on 19th Apr. 2023).

[17] etherscan.io. *Average Daily Transaction Fee*. URL: https://etherscan.io/chart/avg-txfee-usd (visited on 7th Oct. 2022).

[18] etherscan.io. *Ethereum Daily Transactions Chart*. URL: https://etherscan.io/chart/tx (visited on 7th Oct. 2022).

[19] *FAQ: Blockchain & DeFi Basics |RadixDLT*. Sept. 2022. URL: https://learn.radixdlt.com/article/whats-the-difference-between-proof-of-stake-pos-and-delegated-proof-of-stake-dpos (visited on 12th Oct. 2022).

[20] The RADIX FOUNDATION. *Radix DeFi White paper*. Aug. 2020. URL: https://www.radixdlt.com/post/defi-whitepaper-how-radix-is-building-the-future-of-defi (visited on 10th Oct. 2022).

[21] Fredrik Haga. *DEX Tracker - Decentralized Exchanges Trading Volume |defiprime.com*. URL: https://defiprime.com/dex-volume (visited on 3rd Oct. 2022).

[22] Tobias Hagehei. *DefiFunds*. URL: https://github.com/tobben1998/scrypto/tree/SpecializationProject/defi_fund (visited on 19th May 2023).

[23] Tobias Hagehei. *DefiFunds*. URL: https://devpost.com/software/ll-pgqcjd (visited on 2nd June 2023).

[24] Tobias Hagehei. *DefiFunds - A proof of concept Dapp built on Radix*. Dec. 2022. URL: https://github.com/radixdlt/scrypto-challenges/blob/main/7-defi-devpost/DefiFunds/DefiFunds_-_A_proof_of_concept_Dapp_built_on_Radix.pdf (visited on 1st June 2023).

[25] Tobias Hagehei. *DefiFunds on Betanet V2*. URL: https://github.com/tobben1998/scrypto/tree/master/defi_fund (visited on 19th May 2023).

[26] Tobias Hagehei. *DefiFunds on Betanet V2*. URL: https://github.com/radixdlt/scrypto-challenges/tree/main/7-defi-devpost/DefiFunds (visited on 4th June 2023).

[27] Jelle Hellings et al. *Cerberus: Minimalistic Multi-shard Byzantine-resilient Transaction Processing*. 2020. DOI: 10.48550/ARXIV.2008.04450. URL: https://arxiv.org/abs/2008.04450.

[28] *How The Merge impacted ETH supply |Ethereum.org*. Sept. 2022. URL: https://ethereum.org/en/upgrades/merge/issuance/ (visited on 30th Sept. 2022).

[29] Markus Jakobsson and Ari Juels. 'Proofs of Work and Bread Pudding Protocols(Extended Abstract)'. In: *Secure Information Networks*. Springer US, 1999, pp. 258–272. DOI: 10.1007/978-0-387-35568-9_18.

[30] Yuen Lo and Francesca Medda. 'Uniswap and the rise of the decentralized exchange'. In: (2020). URL: https://mpra.ub.uni-muenchen.de/103925/.

[31] Loi Luu et al. 'Making Smart Contracts Smarter'. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Oct. 2016. DOI: 10.1145/2976749.2978309.

[32] Rachel McIntosh. *Is the Binance Smart Chain Centralized? Messari Researchers Raise Concerns*. Apr. 2021. URL: https://www.financemagnates.com/cryptocurrency/news/is-the-binance-smart-chain-centralized-messari-researchers-raise-concerns/ (visited on 7th Oct. 2022).

[33] MetaMask. *Integrate with the MetaMask wallet*. URL: https://docs.metamask.io/wallet (visited on 10th May 2023).

[34] Kirsty Moreland. *Crypto's Greatest Weakness? Blind Signing, Explained |Ledger.com*. 2022. URL: https://www.ledger.com/academy/cryptos-greatest-weakness-blind-signing-explained (visited on 3rd Oct. 2022).

[35] Kirsty Moreland. *The Safest Way to Use MetaMask With Ledger Hardware Wallet |Ledger*. 2022. URL: https://www.ledger.com/academy/security/the-safest-way-to-use-metamask (visited on 7th Oct. 2022).

[36] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Oct. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[37] Jason Scharfman. 'Decentralized Finance (DeFi) Compliance and Operations'. In: *Cryptocurrency Compliance and Operations*. Springer International Publishing, Nov. 2021, pp. 171–186. URL: https://link.springer.com/chapter/10.1007/978-3-030-88000-2_9.

[38] Corwin Smith. *Proof-of-stake (PoS) |Ethereum.org*. Sept. 2022. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/ (visited on 22nd Sept. 2022).

[39] Ali Sunyaev. *Internet computing: Principles of Distributed systems and emerging internet-based technologies*. Springer Nature, 2020. Chap. 9. URL: https://link.springer.com/content/pdf/10.1007/978-3-030-34957-8.pdf.

[40] *Tokens - OpenZeppelin Docs.* URL: https://docs.openzeppelin.com/contracts/4.x/tokens (visited on 28th Nov. 2022).

[41] Nikhil Vadgama, Jiahua Xu and Paolo Tasca. *Enabling the Internet of Value.* URL: https://link.springer.com/content/pdf/10.1007/978-3-030-78184-2.pdf.

[42] RDX Works. *Babylon Betanet is Live.* Dec. 2022. URL: https://www.radixdlt.com/blog/babylon-betanet-is-live (visited on 17th Apr. 2023).

[43] RDX Works. *Babylon RCnet is Live.* Mar. 2023. URL: https://www.radixdlt.com/blog/babylon-rcnet-released (visited on 17th Apr. 2023).

[44] RDX Works. *Cerberus Infographic Series - Chapter I |TheRadixBlog|RadixDLT.* URL: https://www.radixdlt.com/post/cerberus-infographic-series-chapter-i (visited on 30th Nov. 2022).

[45] RDX Works. *FAQ: Radix Overview.* URL: https://learn.radixdlt.com/article/what-is-the-radix-roadmap (visited on 3rd June 2023).

[46] RDX Works. *Gateway API SDK.* URL: https://github.com/radixdlt/babylon-gateway/tree/main/sdk/typescript (visited on 18th May 2023).

[47] RDX Works. *Radiswap.* Sept. 2022. URL: https://github.com/radixdlt/scrypto-examples/tree/main/defi/radiswap (visited on 1st Nov. 2022).

[48] RDX Works. *Radix Babylon Gateway API (0.1.0).* URL: https://betanet-gateway.redoc.ly/ (visited on 17th Apr. 2023).

[49] RDX Works. *Scrypto101.* URL: https://academy.radixdlt.com/course/scrypto101 (visited on 27th Apr. 2023).

[50] RDX Works. *The Gumball Machine on Betanet V2.* URL: https://github.com/radixdlt/scrypto-examples/tree/main/full-stack/dapp-toolkit-gumball-machine (visited on 1st Mar. 2023).

[51] RDX Works. *Using the Radix Wallet.* URL: https://docs-babylon.radixdlt.com/main/getting-started-developers/wallet/wallet-overview.html (visited on 10th May 2023).

[52] RDX Works. *Wallet SDK.* URL: https://github.com/radixdlt/wallet-sdk#readme (visited on 16th May 2023).

[53] RDX Works. *Welcome to the Radix Babylon Tech Docs.* URL: https://docs-babylon.radixdlt.com/ (visited on 27th Apr. 2023).

[54] RDX Works. *What is a dApp on Radix Babylon?* Jan. 2023. URL: https://www.radixdlt.com/blog/what-is-a-dapp (visited on 16th May 2023).

[55] Zibin Zheng et al. 'An overview on smart contracts: Challenges, advances and platforms'. In: *Future Generation Computer Systems* 105 (Apr. 2020), pp. 475–491. DOI: 10.1016/j.future.2019.12.019.

Blueprint code files

This appendix shows the blueprint code files. If you prefer to have the code files you can find them at my submission on the GitHub to Radix [26], alternatively on my GitHub if you want the most updated ones [25].

A.1 Fund

```rust
use scrypto::prelude::*;
use crate::defifunds::*;

external_component! {
    BeakerfiComponentTarget {
        fn swap(
            &mut self,
            input: Bucket,
            output: ResourceAddress
        ) -> Bucket;
    }
}

external_component! {
    BeakerfiPoolComponentTarget {
        fn swap(
            &mut self,
            input: Bucket
        ) -> Bucket;
    }
}

#[blueprint]
mod fund_module{


    struct Fund {
        fund_name: String,
        short_description: String,
        image_link: String,
        website_link: String,
        vaults: HashMap<ResourceAddress, (Vault, Decimal)>, //decimal to get access to vault info. Better apis will come
        fund_manager_badge: ResourceAddress,
        internal_fund_badge: Vault,
        share_token: ResourceAddress,
        total_share_tokens: Decimal,
        fees_fund_manager_vault: Vault,
        deposit_fee_fund_manager: Decimal,
        defifunds: ComponentAddress, //defifunds ComponentAddress to get access to whitelist and defifund deposit fee
    }

    impl Fund {

        pub fn instantiate_fund(
            fund_name: String,
            token: Bucket,
            initial_supply_share_tokens: Decimal,
            deposit_fee_fund_manager: Decimal,
            defifunds: ComponentAddress,
            short_description: String,
            image_link: String,
            website_link: String

        ) -> (ComponentAddress, Bucket, Bucket) {

            let fund_manager_badge: Bucket = ResourceBuilder::new_fungible()
                .divisibility(DIVISIBILITY_NONE)
                .metadata("name", format!("{} manager badge", fund_name))
                .metadata("description", format!("Badge used for managing {}.", fund_name))
                .mint_initial_supply(1);


            let internal_fund_badge: Bucket = ResourceBuilder::new_fungible()
                .divisibility(DIVISIBILITY_NONE)
                .metadata("name", "Internal fund badge")
                .metadata("description", "Badge that has the auhority to mint and burn share tokens.")
                .mint_initial_supply(1);


            let share_tokens: Bucket = ResourceBuilder::new_fungible()
                .divisibility(DIVISIBILITY_MAXIMUM)
                .metadata("name", format!("{} share tokens", fund_name))
                .metadata("description", format!("Tokens used to show what share of {} you have", fund_name))
                .mintable(rule!(require(internal_fund_badge.resource_address())), AccessRule::DenyAll)
                .burnable(rule!(require(internal_fund_badge.resource_address())), AccessRule::DenyAll)
                .mint_initial_supply(initial_supply_share_tokens);


            let access_rules = AccessRules::new()
                .method("change_deposit_fee_fund_manager", rule!(require(fund_manager_badge.resource_address())),
                AccessRule::DenyAll)
                .method("withdraw_collected_fee_fund_manager", rule!(require(fund_manager_badge.resource_address())),
                AccessRule::DenyAll)
                .method("trade_beakerfi", rule!(require(fund_manager_badge.resource_address())),
                AccessRule::DenyAll)
                .method("change_short_description", rule!(require(fund_manager_badge.resource_address())), AccessRule::DenyAll)
                .method("change_image_link", rule!(require(fund_manager_badge.resource_address())), AccessRule::DenyAll)
                .method("change_website_link", rule!(require(fund_manager_badge.resource_address())), AccessRule::DenyAll)
                .default(rule!(allow_all), AccessRule::DenyAll);



            let mut vaults = HashMap::new();
            vaults.insert(token.resource_address(),(Vault::new(token.resource_address()), token.amount())); //new vault to vaults
            vaults.get_mut(&token.resource_address()).unwrap().0.put(token); //putting tokens in the vault


            let mut component = Self {
                fund_name: fund_name,
                short_description: short_description,
                image_link: image_link,
                website_link: website_link,
                fund_manager_badge: fund_manager_badge.resource_address(),
                internal_fund_badge: Vault::with_bucket(internal_fund_badge),
                vaults: vaults,
                total_share_tokens: initial_supply_share_tokens,
                share_token: share_tokens.resource_address(),
                fees_fund_manager_vault: Vault::new(share_tokens.resource_address()),
                deposit_fee_fund_manager: deposit_fee_fund_manager,
                defifunds: defifunds
            }
            .instantiate();
            component.add_access_check(access_rules);

            (component.globalize(),fund_manager_badge, share_tokens)

        }


        ///////////////////
        ///helper method////
        ///////////////////

        fn add_token_to_fund(&mut self, token: Bucket){
            let resource_address=token.resource_address();

            //create a new vault if not exsisting
            if !self.vaults.contains_key(&resource_address){
                let key=resource_address;
                let value=Vault::new(resource_address);
                self.vaults.insert(key,(value, dec!(0)));
            }
            //put token in the vault with specified resource address.
            let value = self.vaults.get_mut(&resource_address).unwrap();
            value.1 += token.amount(); //update field value, because not api for vault amount
            value.0.put(token);
        }


        /////////////////////////
        ///methods for everyone///
        /////////////////////////


        //method for depositing tokens to the fund. You need to deposit each token that exists in the pool.
        //tokens will be taken in the same ratio as the pool has, and the rest of the tokens will be returned back to you.
        pub fn deposit_tokens_to_fund(&mut self, mut tokens: Vec<Bucket>) -> (Bucket, Vec<Bucket>) {

            //calculate min_ratio to find out how much you should take from each bucket,
            //so there is enough to take, an the ratio in the pool remains the same. The rest will be given back
            let mut ratio=tokens[0].amount()/self.vaults.get_mut(&tokens[0].resource_address()).unwrap().0.amount();
            let mut min_ratio=ratio;
            for token in &tokens{
                ratio=token.amount()/(self.vaults.get_mut(&token.resource_address()).unwrap().0.amount());
                if ratio<min_ratio{
                    min_ratio=ratio;
                }
            }

            //take from buckets, and put them into the fund.
            for token in tokens.iter_mut(){
                let amount=min_ratio*(self.vaults.get_mut(&token.resource_address()).unwrap().0.amount());
                self.add_token_to_fund(token.take(amount));
            }

            //mint new sharetokens
            let new_share_tokens=min_ratio*self.total_share_tokens;
            self.total_share_tokens += new_share_tokens;
            let resource_manager = borrow_resource_manager!(self.fees_fund_manager_vault.resource_address());
            let mut share_tokens = self
                .internal_fund_badge
                .authorize(|| resource_manager.mint(new_share_tokens));

            //deposit fee to the fund manager and to defifunds
            let defifunds: DefifundsGlobalComponentRef=self.defifunds.into();

            let fee_fund_manager=(self.deposit_fee_fund_manager/dec!(100))*share_tokens.amount();
            let fee_defifunds=(defifunds.get_defifunds_deposit_fee()/dec!(100))*share_tokens.amount();

            self.fees_fund_manager_vault.put(share_tokens.take(fee_fund_manager));
            defifunds.add_token_to_fee_vaults(share_tokens.take(fee_defifunds));

            (share_tokens, tokens)
        }



        //method that withdraw tokens from the fund relative to how much sharetokens you put into the method.
        pub fn withdraw_tokens_from_fund(&mut self, share_tokens: Bucket) -> Vec<Bucket> {
            assert!(share_tokens.resource_address()==self.fees_fund_manager_vault.resource_address(),
            "Wrong tokens sent. You need to send share tokens.");

            //take fund from vaults and put into a Vec<Bucket> called tokens
            let mut tokens = Vec::new();
            let your_share = share_tokens.amount()/self.total_share_tokens;
            for value in self.vaults.values_mut(){
                let amount=your_share*value.0.amount();
                tokens.push(value.0.take(amount));
                value.1 -= amount; //update field value, because not api for vault amount
            }

            //burn sharetokens
            self.total_share_tokens -= share_tokens.amount();
            let resource_manager = borrow_resource_manager!(self.fees_fund_manager_vault.resource_address());
            self.internal_fund_badge.authorize(|| resource_manager.burn(share_tokens));

            tokens
        }

        //often used in combination with deposit to fund
        pub fn swap_token_for_tokens(&mut self, mut token: Bucket, ratios : Vec<(ResourceAddress, Decimal)>) -> Vec<Bucket>{
            let defifunds: DefifundsGlobalComponentRef=self.defifunds.into();
            let mut dex: BeakerfiComponentTarget = BeakerfiComponentTarget::at(defifunds.get_dex_address());

            let mut buckets = Vec::new();
            let token_amount=token.amount();

            for (i, (address, ratio)) in ratios.iter().enumerate(){
                //if last element swap the rest, to not recive dust becacause of rounding errors with decimal.
                if i==ratios.len()-1{
                    if token.resource_address()==*address{
                        buckets.push(token);
                    } else{
```

```rust
                    buckets.push(dex.swap(token, *address));
                }
                break;
            }
            else{
                let bucket=token.take((*ratio)*token_amount);
                if token.resource_address()==*address{
                    buckets.push(bucket);
                } else{
                    buckets.push(dex.swap(bucket, *address));
                }

            }
        }
        buckets
    }


    //often used in combination with withdraw from fund
    pub fn swap_tokens_for_token(&mut self, tokens: Vec<Bucket>, token_address: ResourceAddress) -> Bucket{
        let defifunds: DefifundsGlobalComponentRef=self.defifunds.into();
        let mut dex: BeakerfiComponentTarget = BeakerfiComponentTarget::at(defifunds.get_dex_address());

        let mut bucket = Bucket::new(token_address);
        for token in tokens{
            if token.resource_address()==token_address{
                bucket.put(token);
            }
            else{
                bucket.put(dex.swap(token, token_address));
            }
        }
        bucket
    }


    /////////////////////////////
    ///methods for fund manager///
    /////////////////////////////


    pub fn withdraw_collected_fee_fund_manager(&mut self) -> Bucket{
        self.fees_fund_manager_vault.take_all()
    }

    pub fn change_deposit_fee_fund_manager(&mut self, new_fee: Decimal){
        assert!(new_fee >= dec!(0) && new_fee <= dec!(5),"Fee need to be in range of 0% to 5%.");
        self.deposit_fee_fund_manager=new_fee;
    }

    //This method lets the fund manager trade with all the funds assests on whitelisted pools.
    //token_address is the asset you want to trade from.
    pub fn trade_beakerfi(&mut self, token_address: ResourceAddress, amount: Decimal, pool_address: ComponentAddress){

        //checks if the pool is whitelisted
        let mut whitelisted=false;
        let defifunds: DefifundsGlobalComponentRef= self.defifunds.into();

        for (&address, &epoch) in defifunds.get_whitelisted_pool_addresses().iter(){
            if address == pool_address && epoch <= Runtime::current_epoch(){
                whitelisted=true;
            }
        }
        assert!(whitelisted, "Trading pool is not yet whitelisted.");

        //do a trade using beakerfi.
        let mut dexpool: BeakerfiPoolComponentTarget = BeakerfiPoolComponentTarget::at(pool_address);

        let bucket_before_swap=self.vaults.get_mut(&token_address).unwrap().0.take(amount);
        self.vaults.get_mut(&token_address).unwrap().1 -= amount; //update field value, because not api for vault amount

        let bucket_after_swap=dexpool.swap(bucket_before_swap);
        self.add_token_to_fund(bucket_after_swap);

    }

    pub fn change_short_description(&mut self, short_description: String){
        self.short_description=short_description;
    }

    pub fn change_image_link(&mut self, image_link: String){
        self.image_link=image_link;
    }

    pub fn change_website_link(&mut self, website_link: String){
        self.website_link=website_link;
        }

    }
}


```

A.2 Defifunds

```
use scrypto::prelude::*;
use crate::fund::*;


#[blueprint]
mod defifunds_module{


    struct Defifunds {
        funds: Vec<(ComponentAddress, ResourceAddress, ResourceAddress)>, //all funds (<fund, fundmanagerbadge, sharetoken>)
        defifunds_admin_badge: ResourceAddress,
        whitelisted_pool_addresses: HashMap<ComponentAddress, u64>, //whitelist valid from epoch <u64>
        defifunds_deposit_fee: Decimal,
        fee_vaults: HashMap<ResourceAddress, Vault>,
        set_component_badge: ResourceAddress, //used for the work around
        component_address: Option<ComponentAddress>, //component address of self. A work around for 0.7.0
        beakerfi: ComponentAddress,
    }

    impl Defifunds {

        pub fn instantiate_defifunds(beakerfi: ComponentAddress) -> (ComponentAddress, Bucket) {

            let defifunds_admin_badge: Bucket = ResourceBuilder::new_fungible()
                .divisibility(DIVISIBILITY_NONE)
                .metadata("name", "defifunds admin badge")
                .metadata("description", "Badge used for admin stuff")
                .mint_initial_supply(1);

            //used for workaround for 0.7.0 to get it selves component address
            let set_component_badge: Bucket = ResourceBuilder::new_fungible()
                .divisibility(DIVISIBILITY_NONE)
                .metadata("name", "set component badge")
                .metadata("description", "used in 0.7.0 because not possible to get it selves component address")
                .burnable(rule!(allow_all), AccessRule::DenyAll)
                .mint_initial_supply(1);

            let access_rules = AccessRules::new()
                .method("new_pool_to_whitelist", rule!(require(defifunds_admin_badge.resource_address())),
                AccessRule::DenyAll)
                .method("remove_pool_from_whitelist", rule!(require(defifunds_admin_badge.resource_address())),
                AccessRule::DenyAll)
                .method("change_deposit_fee_defifunds", rule!(require(defifunds_admin_badge.resource_address())),
                AccessRule::DenyAll)
                .method("withdraw_collected_fee_defifunds", rule!(require(defifunds_admin_badge.resource_address())),
                AccessRule::DenyAll)
                .method("withdraw_collected_fee_defifunds_all", rule!(require(defifunds_admin_badge.resource_address())),
                AccessRule::DenyAll)
                .default(rule!(allow_all), AccessRule::DenyAll);

            let mut component = Self {
                funds: Vec::new(),
                defifunds_admin_badge: defifunds_admin_badge.resource_address(),
                whitelisted_pool_addresses: HashMap::new(),
                defifunds_deposit_fee: dec!(1),
                fee_vaults: HashMap::new(),
                set_component_badge: set_component_badge.resource_address(),
                component_address: None,
                beakerfi: beakerfi
            }
            .instantiate();
            component.add_access_check(access_rules);
            let globalized_component=component.globalize();
            let defifunds_component: DefifundsGlobalComponentRef = globalized_component.into();
            defifunds_component.set_address(globalized_component, set_component_badge); //workaround to get component address.

            (globalized_component, defifunds_admin_badge)

        }

        //helper method for 0.7.0 to set component address. Can only be used when instatiating, because no one get the badge.
        pub fn set_address(&mut self, address: ComponentAddress, badge: Bucket){
            assert_eq!(badge.resource_address(), self.set_component_badge,
```

```rust
            "The badge is only accasable when instantiating a new component, so no need to call this method.");
            self.component_address = Some(address);
            badge.burn();
        }

        //fund make use of this method to deposit the fee to the correct vault
        //if other people decide to use this method it is just free money to the defifunds admin :D
        pub fn add_token_to_fee_vaults(&mut self, token: Bucket){
            let resource_address=token.resource_address();

            if !self.fee_vaults.contains_key(&resource_address){
                let key=resource_address;
                let value=Vault::new(resource_address);
                self.fee_vaults.insert(key,value);
            }

            self.fee_vaults.get_mut(&resource_address).unwrap().put(token);
        }

        /////////////////////////
        ///methods for everyone///
        /////////////////////////

        pub fn new_fund(&mut self,
            fund_name: String,
            token: Bucket,
            initial_supply_share_tokens: Decimal,
            deposit_fee_fund_manager: Decimal,
            short_description: String,
            image_link: String,
            website_link: String
        ) -> (Bucket, Bucket){
            let (fund, fund_manager_badge, share_tokens)=FundComponent::instantiate_fund(
                fund_name,
                token,
                initial_supply_share_tokens,
                deposit_fee_fund_manager,
                self.component_address.unwrap(), //component address of Defifunds
                short_description,
                image_link,
                website_link
            )
            .into();
            self.funds.push((fund.into(),fund_manager_badge.resource_address(),share_tokens.resource_address()));

            (fund_manager_badge, share_tokens)
        }

        pub fn get_funds(&mut self) -> Vec<(ComponentAddress, ResourceAddress, ResourceAddress)>{
            self.funds.clone()
        }

        pub fn get_dex_address(&mut self) -> ComponentAddress{
            self.beakerfi.clone()
        }

        pub fn get_defifunds_deposit_fee(&mut self) -> Decimal{
            self.defifunds_deposit_fee
        }

        pub fn get_whitelisted_pool_addresses(&mut self) -> HashMap<ComponentAddress, u64>{
            self.whitelisted_pool_addresses.clone()
        }



        ////////////////////////////////
        ///methods for defifund admin///
        ////////////////////////////////

        pub fn new_pool_to_whitelist(&mut self, pool_address: ComponentAddress){
            self.whitelisted_pool_addresses.insert(pool_address, Runtime::current_epoch());//+300); //removed while testing.
        }

        pub fn remove_pool_from_whitelist(&mut self, pool_address: ComponentAddress){
            self.whitelisted_pool_addresses.remove(&pool_address);
        }

        pub fn change_deposit_fee_defifunds(&mut self, new_fee: Decimal){
            assert!(new_fee >= dec!(0) && new_fee <= dec!(5),"Fee need to be in range of 0% to 5%.");
            self.defifunds_deposit_fee=new_fee;
        }

        pub fn withdraw_collected_fee_defifunds(&mut self, address: ResourceAddress) -> Bucket{
            self.fee_vaults.get_mut(&address).unwrap().take_all()
        }
        pub fn withdraw_collected_fee_defifunds_all(&mut self) -> Vec<Bucket>{
            let mut tokens = Vec::new();
            for vault in self.fee_vaults.values_mut(){
                tokens.push(vault.take_all());
            }
            tokens
        }

    }
}

```

Javascript code files

This appendix shows the javascript code files for the frontend. The HTML and CSS files are dropped, but you can find them on GitHub. If you prefer to have the code files you can find them at my submission on the GitHub to Radix [26], alternatively on my GitHub if you want the most updated ones [25].

RadixConnect is a file that contains the code related to the connect button and sending manifests to the Betanet. The Index file is the file that contains all the transaction manifests. apiDataFecther contains general functions for doing API calls, saving that data, and retrieving specific data from the saved data. The four other files are specific for each frontend site on the UI design that requires data. RadixConnect and index are written with inspiration from the gumball example made by RDX Works [50].

B.1 radixConnect

```
import {
  RadixDappToolkit,
  ManifestBuilder,
  Decimal,
  Bucket,
  Expression,
  ResourceAddress,
} from "@radixdlt/radix-dapp-toolkit";

// Configure the connect button
export let accountAddress;
const rdt = RadixDappToolkit(
  {
    dAppDefinitionAddress:
      "account_tdx_b_1pplsymjqavhw82vkw69h6zkj5r2gzrh47lvdd0s8h0jseu8sqt",
    dAppName: "defifunds",
  },
  (requestData) => {
    requestData({
      accounts: { quantifier: "atLeast", quantity: 1 },
    }).map(({ data: { accounts } }) => {
      // add accounts to dApp application state
      console.log("account data: ", accounts);
      document.getElementById("accountName").innerText = accounts[0].label;
      document.getElementById("accountAddress").innerText = accounts[0].address;
      accountAddress = accounts[0].address;
    });
  },
  { networkId: 11 }
);
console.log("dApp Toolkit: ", rdt);

// There are four classes exported in the Gateway-SDK These serve as a thin wrapper around the gateway API
// API docs are available @ https://betanet-gateway.redoc.ly/
import {
  TransactionApi,
  StateApi,
  StatusApi,
  StreamApi,
} from "@radixdlt/babylon-gateway-api-sdk";

// Instantiate Gateway SDK
const transactionApi = new TransactionApi();
const stateApi = new StateApi();
const statusApi = new StatusApi();
const streamApi = new StreamApi();

// ************ Send Manifest*************
export async function sendManifest(manifest) {
  // Send manifest to extension for signing
  const result = await rdt.sendTransaction({
    transactionManifest: manifest,
    version: 1,
  });
  if (result.isErr()) throw result.error;
  console.log("Result: ", result.value);

  // Fetch the transaction status from the Gateway API
  let status = await transactionApi.transactionStatus({
    transactionStatusRequest: {
      intent_hash_hex: result.value.transactionIntentHash,
    },
  });
  console.log(" TransactionApi transaction/status:", status);

  // fetch component address from gateway api and set componentAddress variable
  let commitReceipt = await transactionApi.transactionCommittedDetails({
    transactionCommittedDetailsRequest: {
      transaction_identifier: {
        type: "intent_haFsh",
        value_hex: result.value.transactionIntentHash,
      },
    },
  });
  console.log("Committed Details Receipt", commitReceipt);

  return { status, commitReceipt };
}

// ************ Show Recript*************
export function showReceipt(commitReceipt, fieldId) {
  document.getElementById(fieldId).innerText = JSON.stringify(
    commitReceipt.details.receipt,
    null,
    2
  );
}

```

B.2 index

"index.js" is not the cleanest looking code file, but the point was only to test the transaction calls up against the network. The interesting part you need from this code file is the local variables named "manifest".

```
import {
  RadixDappToolkit,
  ManifestBuilder,
  Decimal,
  Array,
  Tuple,
  String,
  Bucket,
  Expression,
  ResourceAddress,
  ComponentAddress,
} from "@radixdlt/radix-dapp-toolkit";

import axios from "axios";
import { accountAddress, sendManifest, showReceipt } from "./radixConnect.js";
import {
  fetchPoolInfo,
  xrdAddr,
  getRatios,
  getTokenPrices,
  getFunds,
  getFundsInfo,
  getTokensInWallet,
  updateAll,
  getFundTokenAmount,
  getTokenAmount,
} from "./apiDataFetcher.js";
import { getPortfolio, getSharetokensWallet } from "./myInvestments";
import {
  getFundManagerFunds,
  getYourShareAndTvl,
  getManageFundPortfolio,
} from "./manageFund.js";
import {
  getFundPortfolio,
} from "./fund.js";

// Global states
export let DefiFundsComponentAddress =
  "component_tdx_b_1qff318hj4le2nppf6yzgn293rj9g7yyu7v079gfq4h5sv60qxz";
let DefiFundsAdminBadge =
  "resource_tdx_b_1qpf318hj4le2nppf6yzgn293rj9g7yyu7v079gfq4h5sxmawre";

let FundComponentAddress;
let FundManagerBadge;
let ShareTokenAddress;

document.getElementById("test").onclick = async function () {
  await updateAll(
    "account_tdx_b_1prtyyczzd3fhmrt39fwhtvel9gjhe0ghmane94sfqhyq57045z"
  );
  const selectedFund =
    "component_tdx_b_1q2cumvgvegl4lt9natpzldcz9wnuhh59pu4md5wdj0msl48ad5";

  console.log(
    "NB! Accountaddress and Fundaddress is hardcoded in the index.js file. Chnage there if you want to see your own."
  );
  console.log("getTokenPrices:", getTokenPrices());
  console.log("getFunds", getFunds());
  console.log("getFundsInfo", getFundsInfo());
  console.log("getTokensInWallet", getTokensInWallet());
  console.log("getSharetokensWallet", getSharetokensWallet());
  console.log("getPortfolio", getPortfolio());
  console.log("getFundManagerFunds", getFundManagerFunds());

  console.log("getYourShareAndTvl", getYourShareAndTvl(selectedFund));
  console.log("getManageFundPortfolio", getManageFundPortfolio(selectedFund));
  console.log("getYourShareAndTvl", getYourShareAndTvl(selectedFund));
  console.log("getFundPortfolio", getFundPortfolio(selectedFund));
  console.log("getFundTokenAmount", getFundTokenAmount(selectedFund, xrdAddr));
  console.log("getTokenAmount", getTokenAmount(xrdAddr));
};

// ************************************
// ************ DefiFunds ************
// ************************************

// ************ Instantiate component and fetch component and resource addresses ************
document.getElementById("instantiateDefiFunds").onclick = async function () {
  let packageAddress = document.getElementById("packageAddress").value;
  let dexComponentAddress = document.getElementById(
    "dexComponentAddress"
  ).value;

  let manifest = new ManifestBuilder()
    .callFunction(packageAddress, "Defifunds", "instantiate_defifunds", [
      ComponentAddress(dexComponentAddress),
    ])
    .callMethod(accountAddress, "deposit_batch", [Expression("ENTIRE_WORKTOP")])
    .build()
    .toString();
  console.log("Manifest: ", manifest);

  const { commitReceipt } = await sendManifest(manifest);

  // set componentAddress variable with gateway api commitReciept payload
  // componentAddress = commitReceipt.details.receipt.state_updates.new_global_entities[0].global_address <- long way
  //shorter way below->
  DefiFundsComponentAddress =
    commitReceipt.details.referenced_global_entities[0];
  DefiFundsAdminBadge = commitReceipt.details.referenced_global_entities[1];
  document.getElementById("DefiFundsComponentAddress").innerText =
    DefiFundsComponentAddress;
  document.getElementById("DefiFundsAdminBadge").innerText =
    DefiFundsAdminBadge;
};

// ************ New Fund ************
document.getElementById("btnNewFund").onclick = async function () {
  let fundName = document.getElementById("inpNewFundName").value;
  let initialSupply = document.getElementById("inpNewFundInitialSupply").value;
  let depositFee = document.getElementById("inpNewFundDepositFee").value;
  let description = document.getElementById("inpNewFundDescription").value;
  let imagelink = document.getElementById("inpNewFundImageLink").value;
  let websitelink = document.getElementById("inpNewFundWebsiteLink").value;
  let manifest = new ManifestBuilder()
    .withdrawFromAccountByAmount(accountAddress, initialSupply, xrdAddr)
    .takeFromWorktopByAmount(initialSupply, xrdAddr, "xrd_bucket")
    .callMethod(DefiFundsComponentAddress, "new_fund", [
      String(fundName),
      Bucket("xrd_bucket"),
      Decimal(initialSupply),
      Decimal(depositFee),
      String(description),
      String(imagelink),
      String(websitelink),
    ])
    .callMethod(accountAddress, "deposit_batch", [Expression("ENTIRE_WORKTOP")])
    .build()
    .toString();

  console.log("Manifest: ", manifest);

  const { commitReceipt } = await sendManifest(manifest);

  document.getElementById("StatusNewFund").innerText =
    commitReceipt.details.receipt.status;
  document.getElementById("FundComponentAddressNewFund").innerText =
    commitReceipt.details.referenced_global_entities[1];
  document.getElementById("FundManagerBadgeNewFund").innerText =
    commitReceipt.details.referenced_global_entities[2];
  document.getElementById("ShareTokenAddressNewFund").innerText =
    commitReceipt.details.referenced_global_entities[4];
};

// ************ New Pool To Whitelist *************
document.getElementById("btnNewPoolToWhitelist").onclick = async function () {
  let pool = document.getElementById("inpNewPoolAddress").value;
  let manifest = new ManifestBuilder()
    .createProofFromAccountByAmount(accountAddress, 1, DefiFundsAdminBadge)
    .callMethod(DefiFundsComponentAddress, "new_pool_to_whitelist", [
      ComponentAddress(pool),
    ])
    .build()
    .toString();

  console.log("Manifest: ", manifest);

  const { commitReceipt } = await sendManifest(manifest);

  document.getElementById("StatusNewPoolToWhitelist").innerText =
    commitReceipt.details.receipt.status;
};

// ************ Remove Pool From Whitelist *************
document.getElementById("btnRemovePoolFromWhitelist").onclick =
  async function () {
    let pool = document.getElementById("inpRemovePoolAddress").value;
    let manifest = new ManifestBuilder()
      .createProofFromAccountByAmount(accountAddress, 1, DefiFundsAdminBadge)
      .callMethod(DefiFundsComponentAddress, "remove_pool_from_whitelist", [
        ComponentAddress(pool),
      ])
      .build()
      .toString();

    console.log("Manifest: ", manifest);

    const { commitReceipt } = await sendManifest(manifest);

    document.getElementById("StatusRemovePoolFromWhitelist").innerText =
      commitReceipt.details.receipt.status;
  };

// ************ Change deposit fee defifunds *************
document.getElementById("btnChangeDepositFeeDefifunds").onclick =
  async function () {
    let new_fee = document.getElementById("inpChangeDepositFeeDefifunds").value;
    let manifest = new ManifestBuilder()
      .createProofFromAccountByAmount(accountAddress, 1, DefiFundsAdminBadge)
      .callMethod(DefiFundsComponentAddress, "change_deposit_fee_defifunds", [
        Decimal(new_fee),
      ])
      .build()
      .toString();

    console.log("Manifest: ", manifest);

    const { commitReceipt } = await sendManifest(manifest);

    document.getElementById("StatusChangeDepositFeeDefifunds").innerText =
      commitReceipt.details.receipt.status;
  };

// ************ Withdraw collected fee defifunds *************
document.getElementById("btnWithdrawCollectedFeeDefifunds").onclick =
  async function () {
    let address = document.getElementById(
      "inpWithdrawCollectedFeeDefifunds"
    ).value;
    let manifest = new ManifestBuilder()
      .createProofFromAccountByAmount(accountAddress, 1, DefiFundsAdminBadge)
      .callMethod(
        DefiFundsComponentAddress,
        "withdraw_collected_fee_defifunds",
        [ResourceAddress(address)]
      )
      .callMethod(accountAddress, "deposit_batch", [
        Expression("ENTIRE_WORKTOP"),
      ])
      .build()
      .toString();

    console.log("Manifest: ", manifest);

    const { commitReceipt } = await sendManifest(manifest);

    document.getElementById("StatusWithdrawCollectedFeeDefifunds").innerText =
      commitReceipt.details.receipt.status;
  };

// ************ Withdraw collected fee defifunds all *************
document.getElementById("btnWithdrawCollectedFeeDefifundsAll").onclick =
  async function () {
    let manifest = new ManifestBuilder()
      .createProofFromAccountByAmount(accountAddress, 1, DefiFundsAdminBadge)
      .callMethod(
        DefiFundsComponentAddress,
        "withdraw_collected_fee_defifunds_all",
        []
      )
      .callMethod(accountAddress, "deposit_batch", [
        Expression("ENTIRE_WORKTOP"),
      ])
      .build()
      .toString();

    console.log("Manifest: ", manifest);

    const { commitReceipt } = await sendManifest(manifest);

    document.getElementById(
      "StatusWithdrawCollectedFeeDefifundsAll"
    ).innerText = commitReceipt.details.receipt.status;
  };

// ***************************************
// ************ Fund ******************
// ***************************************

// ************ Get Fund Addresses *************
document.getElementById("btnGetFundAddresses").onclick = async function () {
  axios
    .post("https://betanet.radixdlt.com/entity/details", {
      address: DefiFundsComponentAddress,
    })
    .then((response) => {
      let vector = response.data.details.state.data_json[0];
      document.getElementById("rcptFunds").innerText = vector
        .map((arr) => arr.join("\n"))
        .join("\n\n");
    });
};

// ************ Set Fund Address *************
document.getElementById("btnSetFundAddress").onclick = async function () {
  FundComponentAddress = document.getElementById("inpSetFundAddress").value;
  FundManagerBadge = document.getElementById("inpSetFundManagerBadge").value;
  ShareTokenAddress = document.getElementById("inpSetShareToken").value;
};

// ************ Get pool info *************
```

```
document.getElementById("btnGetPoolInfo").onclick = async function () {
  let selectElement = document.getElementById("selGetPoolInfo");
  let value = selectElement.options[selectElement.selectedIndex].value;
  let addresses = value.split(",");
  let address1 = addresses[0];
  let address2 = addresses[1];
  let noe = await fetchPoolInfo(address1, address2);
  //let noe = await request_pool_info();
  console.log(noe);
  console.log("Price: ", noe[1] / noe[0]);
};

// ************ Deposit tokens to fund *************
document.getElementById("btnDeposit").onclick = async function () {
  let ratios = await getRatios(FundComponentAddress);
  let ratioTuples = [];
  for (let [address, ratio] of ratios) {
    ratioTuples.push(Tuple(ResourceAddress(address), Decimal(ratio)));
  }
  console.log(ratios);
  let amount = document.getElementById("inpDepositFromNumber").value;
  let selectElement = document.getElementById("selDepositFromAddress");
  let address = selectElement.options[selectElement.selectedIndex].value;
  let manifest = new ManifestBuilder()
    .withdrawFromAccountByAmount(accountAddress, amount, address)
    .takeFromWorktopByAmount(amount, address, "bucket")
    .callMethod(FundComponentAddress, "swap_token_for_tokens", [
      Bucket("bucket"),
      Array("Tuple", ...ratioTuples),
    ])
    .callMethod(FundComponentAddress, "deposit_tokens_to_fund", [
      Expression("ENTIRE_WORKTOP"), //this is a vec of all buckets on worktop
    ])
    .callMethod(accountAddress, "deposit_batch", [Expression("ENTIRE_WORKTOP")])
    .build()
    .toString();

  console.log("Manifest: ", manifest);

  const { commitReceipt } = await sendManifest(manifest);

  document.getElementById("StatusDeposit").innerText =
    commitReceipt.details.receipt.status;
};

// ************ Withdraw tokens from fund *************
document.getElementById("btnWithdraw").onclick = async function () {
  let amount = document.getElementById("inpWithdrawFromNumber").value;
  let selectElement = document.getElementById("selWithdrawToAddress");
  let address = selectElement.options[selectElement.selectedIndex].value;

  let manifest = new ManifestBuilder()
    .withdrawFromAccountByAmount(accountAddress, amount, ShareTokenAddress)
    .takeFromWorktop(ShareTokenAddress, "bucket")
    .callMethod(FundComponentAddress, "withdraw_tokens_from_fund", [
      Bucket("bucket"),
    ])
    .callMethod(FundComponentAddress, "swap_tokens_for_token", [
      Expression("ENTIRE_WORKTOP"),
      ResourceAddress(address),
    ])
    .callMethod(accountAddress, "deposit_batch", [Expression("ENTIRE_WORKTOP")])
    .build()
    .toString();

  console.log("Manifest: ", manifest);

  const { commitReceipt } = await sendManifest(manifest);

  document.getElementById("StatusWithdraw").innerText =
    commitReceipt.details.receipt.status;
};

// ************ Withdraw collected fee Fund Manager *************
document.getElementById("btnWithdrawCollectedFeeFundManager").onclick =
  async function () {
    let manifest = new ManifestBuilder()
      .createProofFromAccountByAmount(accountAddress, 1, FundManagerBadge)
      .callMethod(
        FundComponentAddress,
        "withdraw_collected_fee_fund_manager",
        []
      )
      .callMethod(accountAddress, "deposit_batch", [
        Expression("ENTIRE_WORKTOP"),
      ])
      .build()
      .toString();

    console.log("Manifest: ", manifest);

    const { commitReceipt } = await sendManifest(manifest);

    document.getElementById("StatusWithdrawCollectedFeeFundManager").innerText =
      commitReceipt.details.receipt.status;
  };

// ************ Change Deposit fee fundmanager *************
document.getElementById("btnChangeDepositFeeFundManager").onclick =
  async function () {
    let newFee = document.getElementById("inpChangeDepositFundManager").value;
    let manifest = new ManifestBuilder()
      .createProofFromAccountByAmount(accountAddress, 1, FundManagerBadge)
      .callMethod(FundComponentAddress, "change_deposit_fee_fund_manager", [
        Decimal(newFee),
      ])
      .build()
      .toString();

    console.log("Manifest: ", manifest);

    const { commitReceipt } = await sendManifest(manifest);

    document.getElementById("StatusChangeDepositFeeFundManager").innerText =
      commitReceipt.details.receipt.status;
  };

// ************ Change Description *************
document.getElementById("btnChangeDescription").onclick = async function () {
  let text = document.getElementById("inpChangeDescription").value;
  let manifest = new ManifestBuilder()
    .createProofFromAccountByAmount(accountAddress, 1, FundManagerBadge)
    .callMethod(FundComponentAddress, "change_short_description", [`"${text}"`])
    .build()
    .toString();

  console.log("Manifest: ", manifest);

  const { commitReceipt } = await sendManifest(manifest);

  document.getElementById("StatusChangeDescription").innerText =
    commitReceipt.details.receipt.status;
};

// ************ Change Image *************
document.getElementById("btnChangeImage").onclick = async function () {
  let text = document.getElementById("inpChangeImage").value;
  let manifest = new ManifestBuilder()
    .createProofFromAccountByAmount(accountAddress, 1, FundManagerBadge)
    .callMethod(FundComponentAddress, "change_image_link", [`"${text}"`])
    .build()
    .toString();

  console.log("Manifest: ", manifest);

  const { commitReceipt } = await sendManifest(manifest);

  document.getElementById("StatusChangeImage").innerText =
    commitReceipt.details.receipt.status;
};

// ************ Change Website *************
document.getElementById("btnChangeWebsite").onclick = async function () {
  let text = document.getElementById("inpChangeWebsite").value;
  let manifest = new ManifestBuilder()
    .createProofFromAccountByAmount(accountAddress, 1, FundManagerBadge)
    .callMethod(FundComponentAddress, "change_website_link", [`"${text}"`])
    .build()
    .toString();

  console.log("Manifest: ", manifest);

  const { commitReceipt } = await sendManifest(manifest);

  document.getElementById("StatusChangeWebsite").innerText =
    commitReceipt.details.receipt.status;
};

//remeber to whitelist the pool before testing

//pool
//xrd
```

```javascript
// ************ Trade Beakerfi *************
document.getElementById("btnTrade").onclick = async function () {
  let amount = document.getElementById("inpTradeAmount").value;
  let selectElement1 = document.getElementById("selTradeFromAddress");
  let address = selectElement1.options[selectElement1.selectedIndex].value;
  let selectElement2 = document.getElementById("selTradeComponentAddress");
  let componentAddress =
    selectElement2.options[selectElement2.selectedIndex].value;

  let manifest = new ManifestBuilder()
    .createProofFromAccountByAmount(accountAddress, 1, FundManagerBadge)
    .callMethod(FundComponentAddress, "trade_beakerfi", [
      ResourceAddress(address),
      Decimal(amount),
      ComponentAddress(componentAddress),
    ])
    .callMethod(accountAddress, "deposit_batch", [Expression("ENTIRE_WORKTOP")])
    .build()
    .toString();

  console.log("Manifest: ", manifest);

  const { commitReceipt } = await sendManifest(manifest);

  document.getElementById("StatusTrade").innerText =
    commitReceipt.details.receipt.status;
};

```

B.3 apiDataFecther

As I am not a frontend developer, I wasn't fully aware of the best practices to limit API calls to what's necessary. I decided that handling the caching or storing of API calls would be better suited for a frontend developer. My priority has been to identify and locate the required data. I've arranged my structure as follows:

- Fetch functions: These are direct API calls.
- Update functions: These leverage the fetch functions to collect and store the fetched data in global variables.
- Get functions: These are used to obtain specific data required by the frontend. If necessary, they can perform calculations on the stored data.

```javascript
import axios from "axios";
import { DefiFundsComponentAddress } from "./index.js";

export const xrdAddr =
  "resource_tdx_b_1qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq8z96qp";

export const tokensInfo = new Map([
  [
    "resource_tdx_b_1qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq8z96qp",
    { name: "Radix", ticker: "XRD", image: "https://example.com/token1.png" },
  ],
  [
    "resource_tdx_b_1qpev6f8v2su68ak5p2fswd6gqml3u7q0lkrtfx99c4ts3zxlah",
    {
      name: "Beta Usd",
      ticker: "BUSD",
      image: "https://example.com/token2.png",
    },
  ],
  [
    "resource_tdx_b_1qps68awewmwmz0az7cxd86l7xhq6v3pez355wq8gra3qw2v7kp",
    {
      name: "Wrapped Ether",
      ticker: "WETH",
      image: "https://example.com/token3.png",
    },
  ],
  [
    "resource_tdx_b_1qre9sv98scqut4k9g3j6kxuvscczv0lzumefwgwhuf6qdu4c3r",
    {
      name: "Wrapped Bitcoin",
      ticker: "WBTC",
      image: "https://example.com/token2.png",
    },
  ],
]);

// all funds in defifunds array each element consists off (fundAddr, shartokenAddr, fundManagerBage)
let funds = [];

//info on funds you have put into here. how updated the data is depends on when you last
//updated the fund info on that speciific fund. map (fundAddress, data_json)
let fundsInfo = new Map();

//All fungible tokens in your wallet
let tokensInWallet = new Map();

//tokenPrices in usd. Prices on those token addresses defined in addr
let tokenPrices = new Map();

///////////////////////////////////////
///////////// API Calls /////////////
///////////////////////////////////////

//NB! you will it most cases call the updatefunctions, and the getfunctions not these fetch functions.

export async function fetchPoolInfo(tokenX, tokenY) {
  const apiUrl = "https://beaker.fi:8888/pool_info_price?";
  const params = `token_x=${tokenX}&token_y=${tokenY}`;

  try {
    const response = await axios.get(apiUrl + params);
    const poolInfo = response.data;
    return poolInfo;
  } catch (error) {
    console.error(error);
  }
}

export async function fetchRadixPrice() {
  return axios
    .get(
      "https://api.coingecko.com/api/v3/simple/price?ids=radix&vs_currencies=usd"
    )
    .then((response) => {
      const price = response.data.radix.usd;
      return price;
    })
    .catch((error) => {
      console.error(error);
    });
}

export async function fetchFundInfo(fundAddress) {
  return axios
    .post("https://betanet.radixdlt.com/entity/details", {
      address: fundAddress,
    })
    .then((response) => {
      return response.data.details.state.data_json;
    });
}

export async function fetchFunds() {
  return axios
    .post("https://betanet.radixdlt.com/entity/details", {
      address: DefiFundsComponentAddress,
    })
    .then((response) => {
      return response.data.details.state.data_json[0];
    });
}

export async function fetchFungibleTokens(address) {
  return axios
    .post("https://betanet.radixdlt.com/entity/fungibles", {
      address: address,
    })
    .then((response) => {
      let vector = response.data.fungibles.items;
      const tokenBalances = new Map();
      for (const item of vector) {
        const tokenAddress = item.address;
        const tokenAmount = parseFloat(item.amount.value);
        tokenBalances.set(tokenAddress, tokenAmount);
      }
      return tokenBalances;
    });
}

export async function fetchAllTokenPricesXrd() {
  // fetch all prices in paralell
  const promises = [];
  tokensInfo.forEach((tokenInfo, tokenAddress) => {
    if (tokenAddress !== xrdAddr) {
      const promise = fetchPoolInfo(tokenAddress, xrdAddr).then((data) => ({
        tokenAddress,
        price: data[1] / data[0],
      }));
      promises.push(promise);
    }
  });
  const results = await Promise.all(promises);
  const prices = {};
  results.forEach(({ tokenAddress, price }) => {
    prices[tokenAddress] = price;
  });
  prices[xrdAddr] = 1;
  return prices;
}

//////////////////////////////////////////////////
///////////// Update global variables /////////
//////////////////////////////////////////////////

//NB! call these functions before you use the get functions

//in usd
export async function updateTokenPrices() {
  const [xrdPrice, prices] = await Promise.all([
    fetchRadixPrice(),
    fetchAllTokenPricesXrd(),
  ]);
  for (const [tokenAddress, price] of Object.entries(prices)) {
    tokenPrices.set(tokenAddress, price * xrdPrice);
  }
}

export async function updateFunds() {
  const f = await fetchFunds();
  funds = f;
}

//input: vec<fundAddr>
export async function updateFundsInfo(funds) {
  const promises = funds.map((fundAddr) => fetchFundInfo(fundAddr));
  const results = await Promise.all(promises);

  for (let i = 0; i < funds.length; i++) {
    fundsInfo.set(funds[i], results[i]);
  }
}

export async function updateTokensInWallet(address) {
  const tokens = await fetchFungibleTokens(address);
  tokensInWallet = tokens;
}

export async function updateAll(walletAddr) {
  await updateFunds();
  const funds = getFunds();
  const fundAddresses = funds.map((fund) => fund[0]);
  const promises = [
    updateFundsInfo(fundAddresses),
    updateTokenPrices(),
    updateTokensInWallet(walletAddr),
  ];
  await Promise.all(promises);
}

////////////////////////////////////
///////////// Get stuff /////////
////////////////////////////////////

//NB! you need to call update function before you will get stuff from these functions

export function getTokenPrices() {
  return tokenPrices;
}

export function getFunds() {
  return funds;
}
```

```
export function getFundsInfo() {
  return fundsInfo;
}

export function getTokensInWallet() {
  return tokensInWallet;
}

export function getTokenPrice(tokenAddress) {
  return tokenPrices.get(tokenAddress) || null;
}

export function getFundInfo(fundAddress) {
  return fundsInfo.get(fundAddress) || null;
}

export function getTokenAmount(tokenAddress) {
  const tokensInWallet = getTokensInWallet(tokenAddress);
  const tokenAmount = tokensInWallet.get(tokenAddress) ?? 0;
  return tokenAmount;
}

export function getFundName(fundAddr) {
  return getFundInfo(fundAddr)[0];
}

export function getFundStrategy(fundAddr) {
  return getFundInfo(fundAddr)[1];
}

export function getFundImage(fundAddr) {
  return getFundInfo(fundAddr)[2];
}

export function getFundWebsite(fundAddr) {
  return getFundInfo(fundAddr)[3];
}

//will likly do this another way when rc net is coming. can check resource amount in vault directly
export function getFundAmounts(fundAddr) {
  const fundAmounts = getFundInfo(fundAddr)[4];
  let map = new Map();
  for (let e of fundAmounts) {
    map.set(e[0], e[1][1]);
  }
  return map;
}

export function getFundManagerBadge(fundAddr) {
  return getFundInfo(fundAddr)[5];
}

export function getShareTokenAddress(fundAddr) {
  return getFundInfo(fundAddr)[7];
}

//will likly do this another way when rc net is coming. amount directly then
export function getShareTokenAmount(fundAddr) {
  return getFundInfo(fundAddr)[8];
}

export function getDepositFee(fundAddr) {
  return getFundInfo(fundAddr)[10];
}

export function getFundTokenAmount(fundAddr, tokenAddr) {
  const fundAmounts = getFundAmounts(fundAddr);
  const tokenAmount = fundAmounts.get(tokenAddr) ?? 0;
  return tokenAmount;
}

export function getFundTvl(FundAddress) {
  const fundAmounts = getFundAmounts(FundAddress);
  let totalValue = 0;

  for (let [tokenAddress, amount] of fundAmounts) {
    const price = getTokenPrice(tokenAddress);
    const value = price * amount;
    totalValue += value;
  }
  return totalValue;
}

export function getFundPrice(FundAddress) {
  const tvl = getFundTvl(FundAddress);
  const amount = getShareTokenAmount(FundAddress);
  return tvl / amount;
}

export function getAllFundAmounts(funds) {
  const amounts = funds.map((fund) => {
    const fundAmounts = getFundAmounts(fund);
    return [fund, fundAmounts];
  });

  return new Map(amounts);
}

export function getAllShareTokenAmounts(funds) {
  const amounts = funds.map((fund) => {
    const shareTokenAmount = getShareTokenAmount(fund);
    return [fund, shareTokenAmount];
  });

  return new Map(amounts);
}

export function getAllFundTvls(funds) {
  const tvls = funds.map((fund) => {
    const fundTvl = getFundTvl(fund);
    return [fund, fundTvl];
  });

  return new Map(tvls);
}

export function getAllFundPrices(funds) {
  const prices = funds.map((fund) => {
    const fundPrice = getFundPrice(fund);
    return [fund, fundPrice];
  });

  return new Map(prices);
}

export async function getRatios(FundAddress) {
  //important to have as new info as possible here, since you use it for smart contratc integration.
  //no dangers, if not, but will get bigger rest amounts.
  await Promise.all([updateTokenPrices(), updateFundsInfo([FundAddress])]);
  const amounts = getFundAmounts(FundAddress);
  let totalValue = 0;
  let values = new Map();

  for (let [address, amount] of amounts.entries()) {
    let value = amount * getTokenPrice(address);
    values.set(address, value);
    totalValue += value;
  }
  let ratios = new Map();
  for (let [address, value] of values.entries()) {
    let ratio = value / totalValue;
    ratios.set(address, ratio);
  }
  return ratios;
}

```

B.4 discoverFund

```
import { getFundTvl, getFunds, getFundImage } from "./apiDataFetcher";

//NB! call updatefunctions in apiDataFecther before you use these

export async function getFundsSortedTvl() {
  const funds = getFunds();
  const tvls = [];

  for (const fund of funds) {
    const fundAddr = fund[0];
    const tvl = getFundTvl(fundAddr);
    tvls.push([fundAddr, getFundName(fundAddr), getFundImage(fundAddr), tvl]);
  }

  const sortedTvls = tvls.sort((a, b) => b[3] - a[3]);
  return sortedTvls;
}

```

B.5 fund

```
import {
  getFundTvl,
  getShareTokenAddress,
  getShareTokenAmount,
  getTokensInWallet,
  getFundAmounts,
  getTokenPrice,
  tokensInfo,
} from "./apiDataFetcher";

//NB! call updatefunctions in apiDataFecther before you use these

export function getYourShareAndTvl(fundAddress) {
  const sharetokenAddress = getShareTokenAddress(fundAddress);
  const amount = getShareTokenAmount(fundAddress);
  const tokenBalances = getTokensInWallet();
  const tvl = getFundTvl(fundAddress);
  const yourAmount = tokenBalances.get(sharetokenAddress) || 0;
  const yourShare = (yourAmount * tvl) / amount;
  return [yourShare, tvl];
}

export function getFundPortfolio(fundAddress) {
  const fundAmounts = getFundAmounts(fundAddress);
  let totalUsdValue = 0;

  const portfolio = new Map();
  for (const [tokenAddress, amount] of fundAmounts) {
    const usdValue = getTokenPrice(tokenAddress) * amount;
    totalUsdValue += usdValue;
    portfolio.set(tokenAddress, usdValue);
  }

  for (const [tokenAddress, usdValue] of portfolio) {
    const percentage = (usdValue / totalUsdValue) * 100;
    const { name, ticker, image } = tokensInfo.get(tokenAddress);
    portfolio.set(tokenAddress, { name, ticker, image, percentage });
  }

  return portfolio;
}

```

B.6 myInvestments

```
import {
  getTokensInWallet,
  getFunds,
  getFundPrice,
  getFundName,
  getFundImage,
} from "./apiDataFetcher";

//NB! call updatefunctions in apiDataFecther before you use these

//get all sharetokens that you have in wallet
export function getSharetokensWallet() {
  const funds = getFunds();
  const tokenBalances = getTokensInWallet();
  const matchingTokens = [];

  for (const [tokenAddress, tokenAmount] of tokenBalances) {
    const matchingFund = funds.find((fund) => fund[2] === tokenAddress);
    if (matchingFund) {
      matchingTokens.push([tokenAddress, tokenAmount, matchingFund[0]]);
    }
  }

  return matchingTokens;
}

//get all info for the my investments page, except history of the fund prices.
export function getPortfolio() {
  const myfunds = getSharetokensWallet();
  const portfolio = new Map();
  let totalUsdValue = 0;

  for (const fund of myfunds) {
    const shareTokenAddress = fund[0];
    const amount = fund[1];
    const fundAddress = fund[2];
    const fundName = getFundName(fundAddress);
    const imageLink = getFundImage(fundAddress);
    const price = getFundPrice(fundAddress);
    const usdValue = price * amount;
    const percentage = 0; //updates later

    portfolio.set(fundAddress, {
      shareTokenAddress,
      fundName,
      imageLink,
      amount,
      usdValue,
      percentage,
    });

    totalUsdValue += usdValue;
  }

  for (const [fundAddress, data] of portfolio) {
    const percentage = (data.usdValue / totalUsdValue) * 100;
    data.percentage = percentage;
    portfolio.set(fundAddress, data);
  }

  return [portfolio, totalUsdValue];
}

```

B.7 manageFund

```
import {
  tokensInfo,
  getFundAmounts,
  getFundTvl,
  getFunds,
  getTokensInWallet,
  getShareTokenAddress,
  getShareTokenAmount,
  getTokenPrice,
  getFundName,
  getFundImage,
} from "./apiDataFetcher";

//NB! call updatefunctions in apiDataFecther before you use these

//returns the funds that logged in user (if you have updatetd tokens in wallet) are fundmanagers for.
export function getFundManagerFunds() {
  const funds = getFunds();
  const tokens = getTokensInWallet();
  const matchingFunds = funds.filter((fund) => tokens.has(fund[1]));
  const fundInfo = new Map();
  for (const fund of matchingFunds) {
    const fundAddr = fund[0];
    const fundManagerBage = fund[1];
    fundInfo.set(fundAddr, [
      getFundName(fundAddr),
      getFundImage(fundAddr),
      fundManagerBage,
    ]);
  }
  return fundInfo;
}

export function getYourShareAndTvl(fundAddress) {
  const shareTokenAddress = getShareTokenAddress(fundAddress);
  const amount = getShareTokenAmount(fundAddress);
  const tokenBalances = getTokensInWallet();
  const tvl = getFundTvl(fundAddress);
  const yourAmount = tokenBalances.get(shareTokenAddress) || 0;
  const yourShare = (yourAmount * tvl) / amount;
  return [yourShare, tvl];
}

export function getManageFundPortfolio(fundAddress) {
  const fundAmounts = getFundAmounts(fundAddress);
  let totalUsdValue = 0;

  const portfolio = new Map();
  for (const [tokenAddress, amount] of fundAmounts) {
    const usdValue = getTokenPrice(tokenAddress) * amount;
    const { name, ticker, image } = tokensInfo.get(tokenAddress);

    totalUsdValue += usdValue;

    portfolio.set(tokenAddress, {
      name,
      ticker,
      image,
      amount,
      usdValue,
    });
  }

  for (const [tokenAddress, data] of portfolio) {
    const percentage = (data.usdValue / totalUsdValue) * 100;
    data.percentage = percentage;
    portfolio.set(tokenAddress, data);
  }

  return portfolio;
}

```