Plutus: Learning a smart contract language

IOG

Plutus: Learning a smart contract language

This book was prepared by the IOG Education team. The majority of the text is taken from the 4rd Plutus Pioneer program that was presented by Lars Brünjes and his colleagues. All program code in this book is taken from the Plutus Pioneer program and is freely available at:

https://github.com/input-output-hk/plutus-pioneer-program/tree/fourth-iteration

Table of Contents

1 Plutus introduction

Plutus is the native smart contract language for Cardano. It is a Turing-complete language written mostly in Haskell, and Plutus smart contracts are effectively Haskell programs. By using Plutus, you can be confident in the correct execution of your smart contracts. It draws from modern language research to provide a safe, full-stack programming environment based on Haskell, the leading purely functional programming language. [1]

So, in order to understand the code in this book, you must understand the basics of the Haskell programming language. The Haskell project contains a list of learning resources as books and tutorials under its Documentation section, from which some are free and some commercial. It can be found at Haskell's official web page:

- https://www.haskell.org/documentation/

IOG provides their own Haskell course that is tailored to the needs of also learning Marlowe, a low code smart contract language and alternative to Plutus. This course can be found at the IOG GitHub page:

https://github.com/input-output-hk/haskell-course

If you would like to learn more about Plutus and its software development kit (SDK) in addition to what this book offers, you can check out the online documentation which can be found at:

- https://plutus.readthedocs.io/en/latest/
- https://plutus-apps.readthedocs.io/en/latest/

1.1 Setting up your development environment

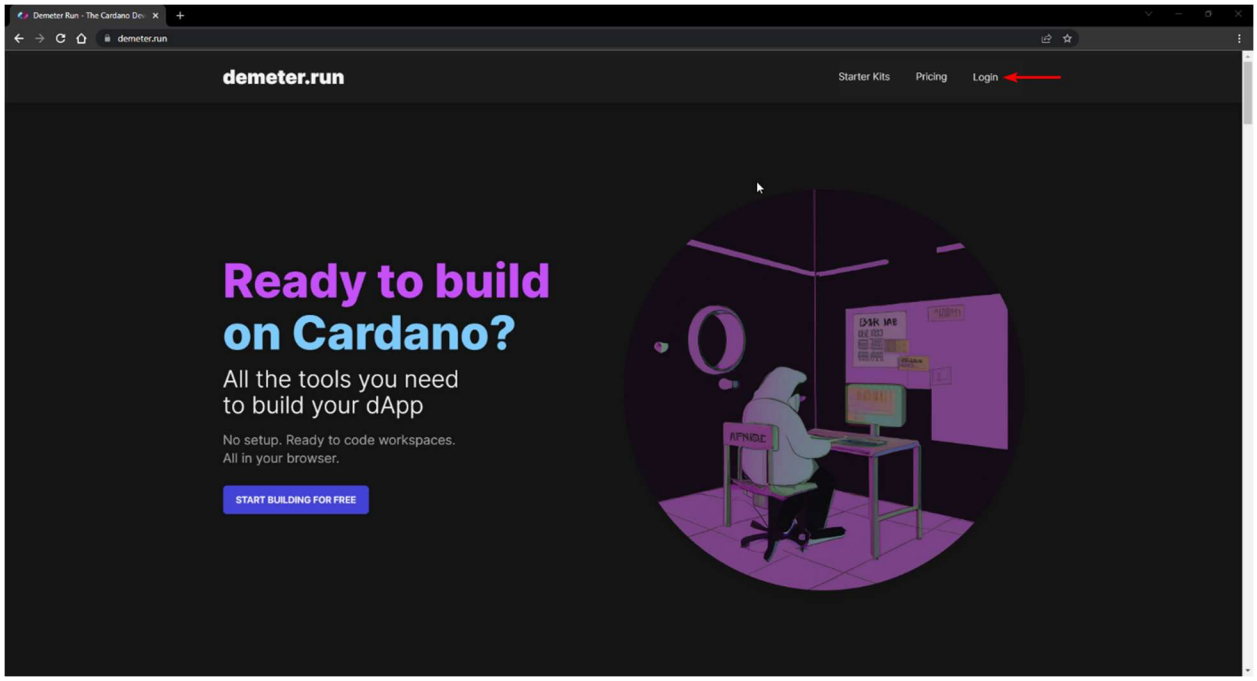
The installation instructions for setting your development environment are split into two parts. The first part is using the online platform *Demeter.run* and the second part is setting up a Docker container locally on your PC. The second instructions are general and can be used on Windows, Mac OS and Linux operating systems.

1.1.1 Using Demeter.run

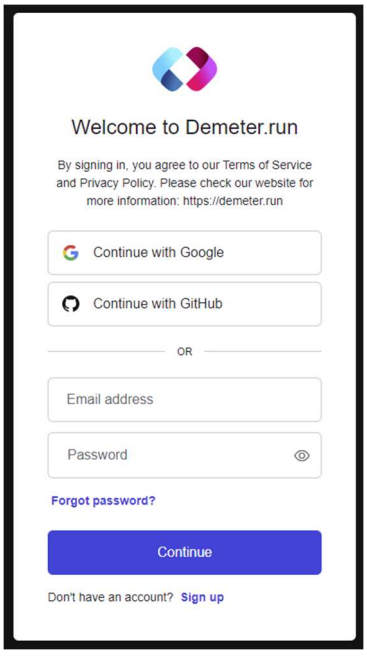
Demeter.run is a cloud-based environment that provides all the tools required for building and deploying Cardano applications. It can be found at https://demeter.run/. To use demeter.run, you need a web browser on any operating system and internet access. Follow the steps below to set up your demeter.run account and to use this development environment.

To setup a demeter.run account and start using it, follow these steps:

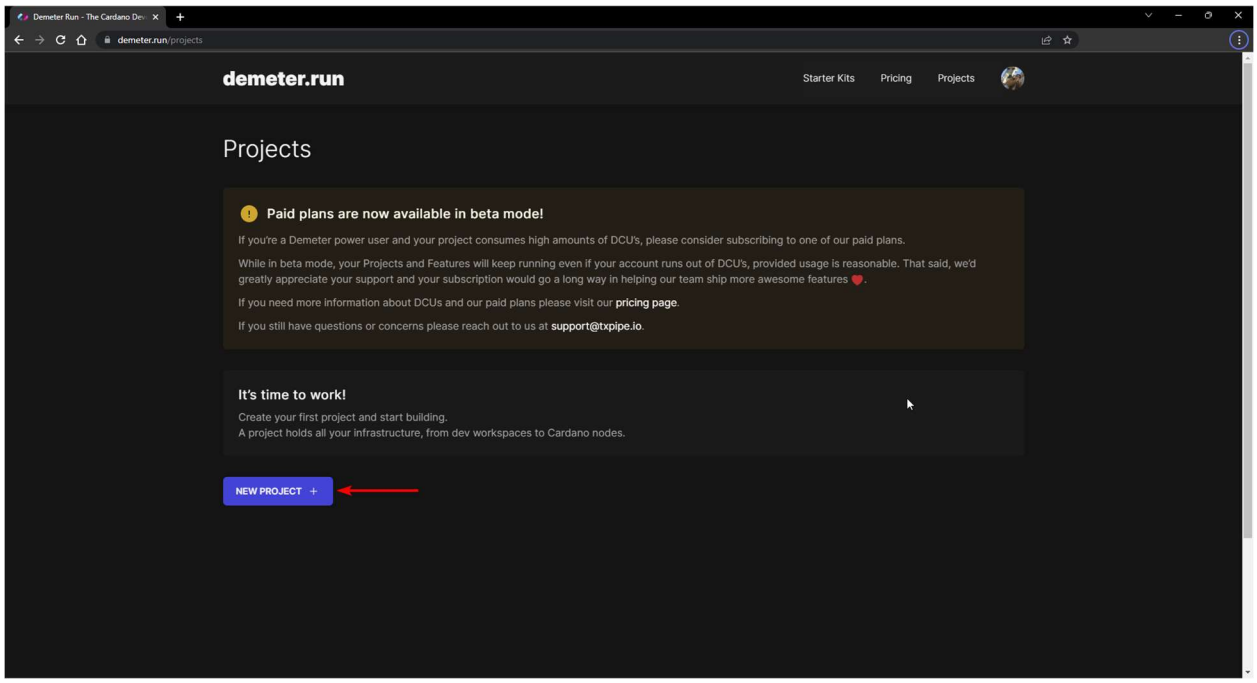
1. Open your browser and navigate to https://demeter.run/. You will see the home page of demeter.run as illustrated below.



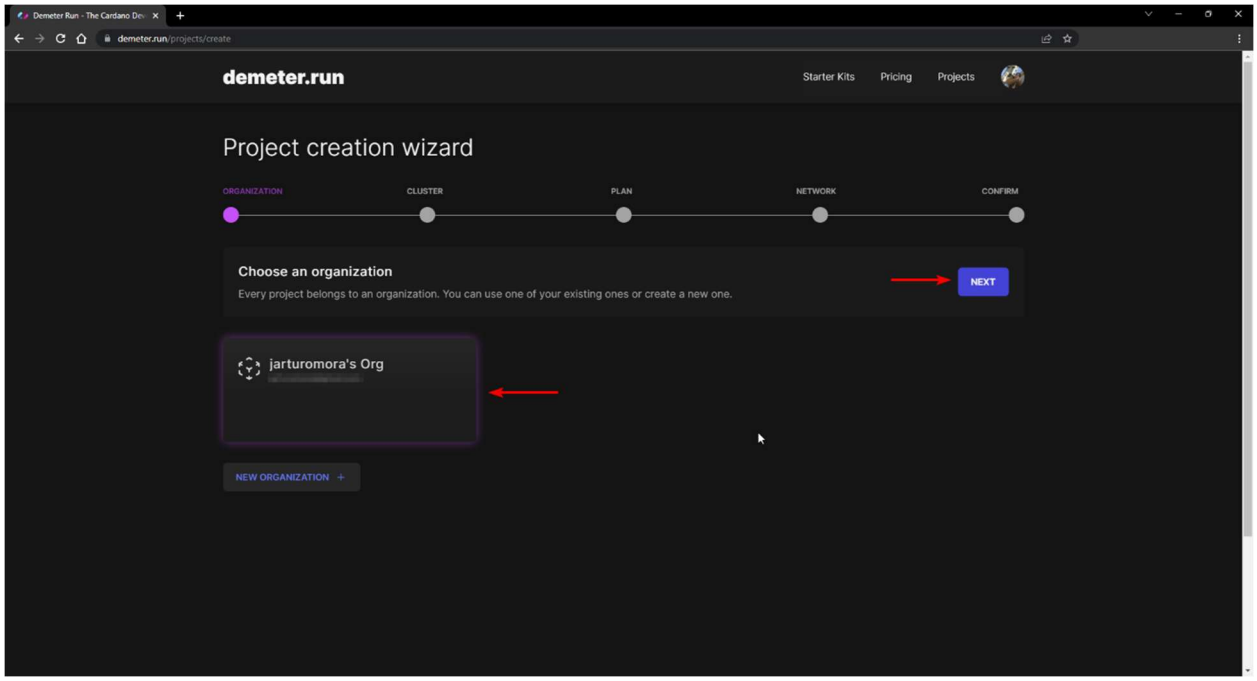
2. Click the **Login** option on the top navigation menu.
3. Next, you need to set up an account. You can choose to set a username and password, or you can sign in by using your Google or GitHub account. Select the method that best fits your preferences to continue.



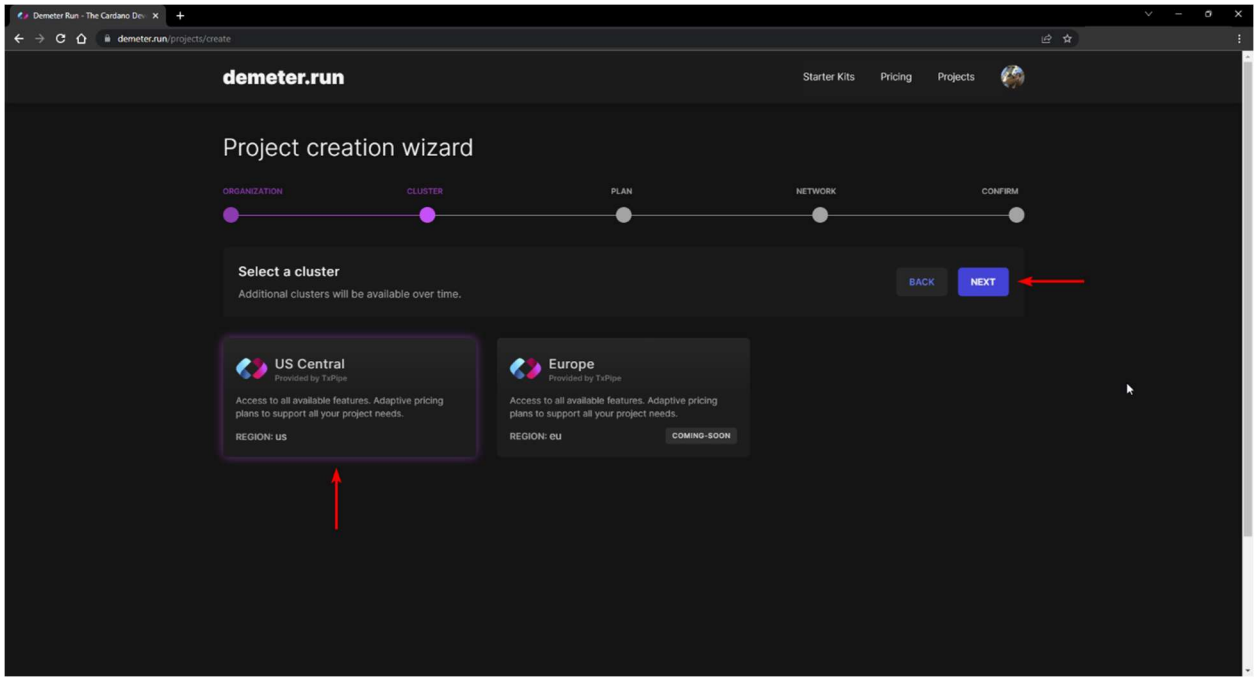
4. After signing in, you are now ready to use demeter.run. You will see the Projects page, as illustrated bellow.



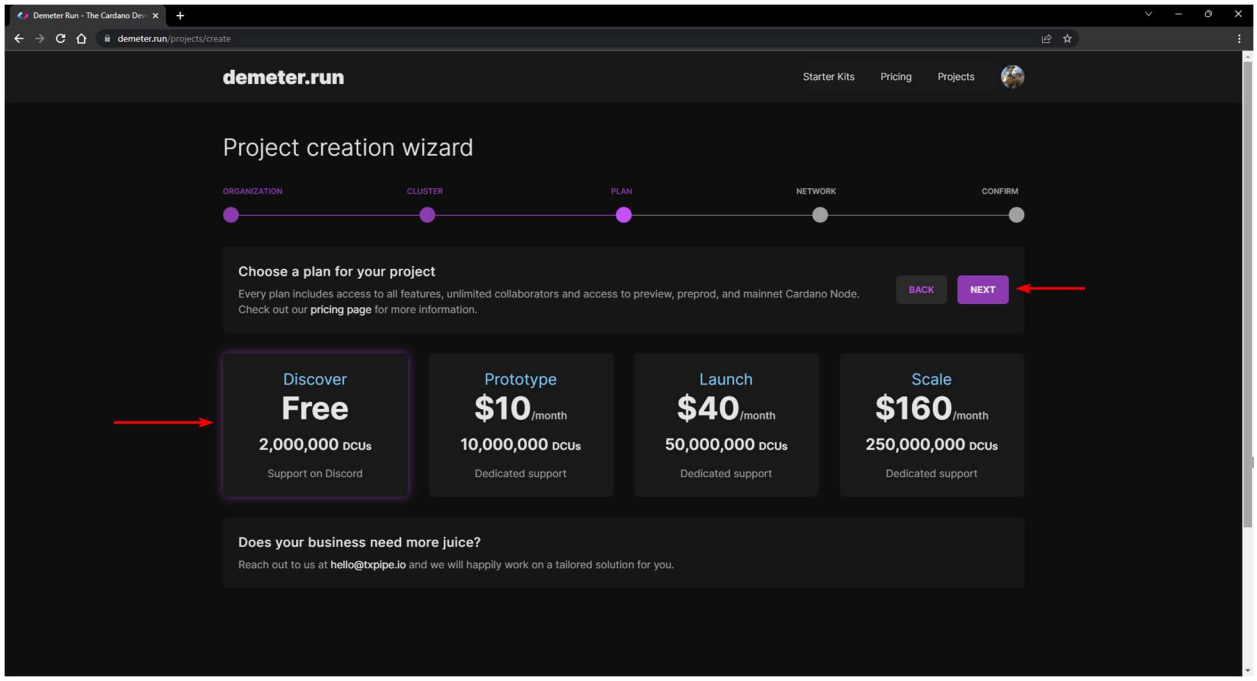
5. Click the **NEW PROJECT** button.
6. Next, you need to follow the required steps to set up a new project. First choose an organization where your project will reside. By default, an organization with your username exists. We will select the default organization for this demo, as shown in the image below, where the username is “jarturomora”.



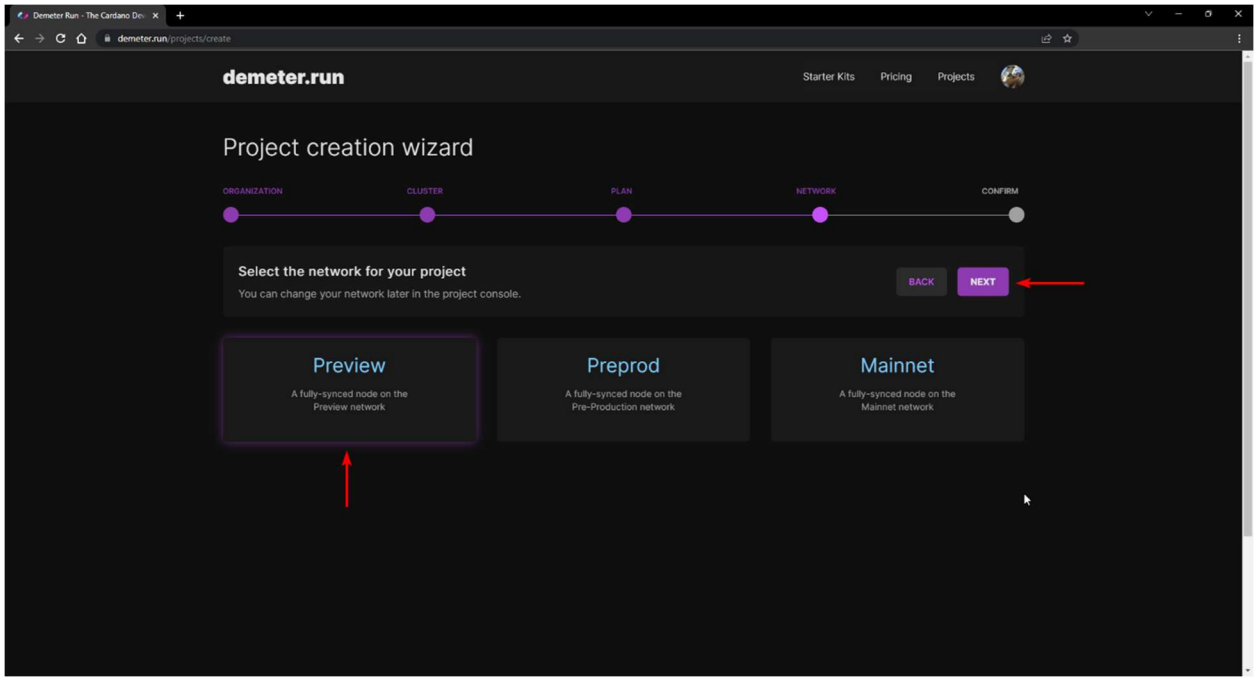
7. After choosing the organization, click the **NEXT** button.
8. Choose the location of the cluster that you will use. As for now, only a US-based cluster exists. Select the **US Central** cluster and click the **NEXT** button to continue, as illustrated below.



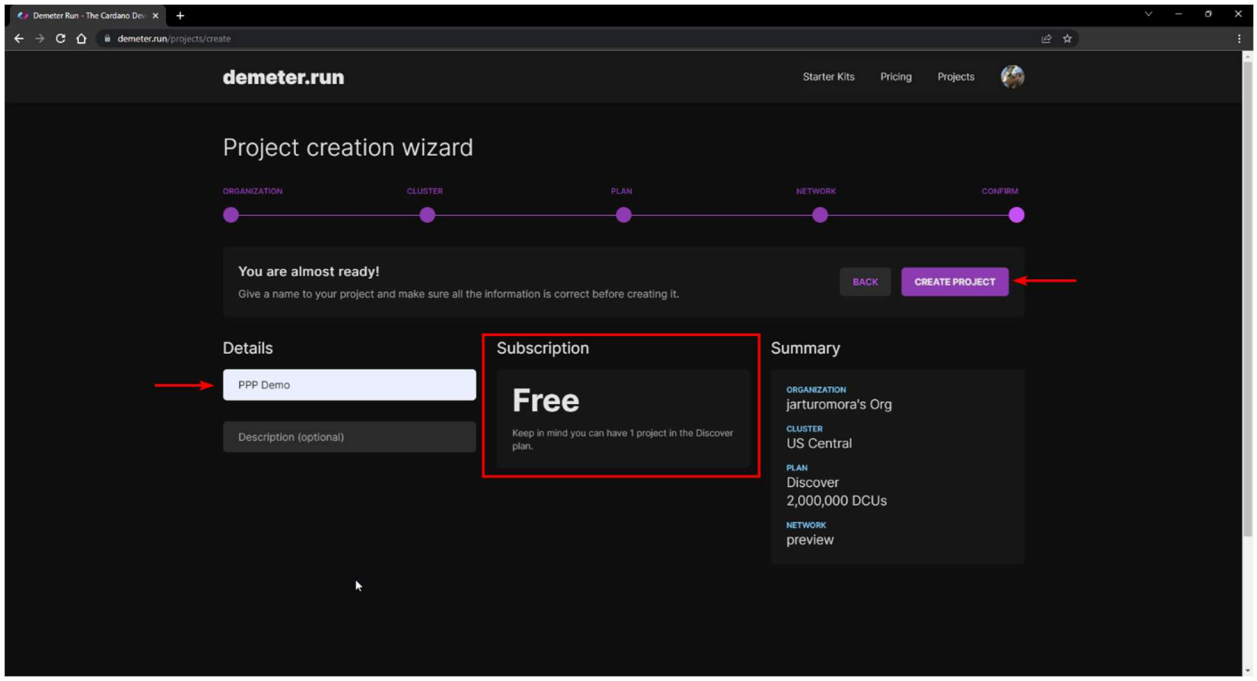
9. After choosing a cluster, the next step is to select a plan. For the purpose of the program, we will use the **Discover** plan that is available free of charge. Select the **Discover** plan and click on the **NEXT** button to continue, as illustrated below.



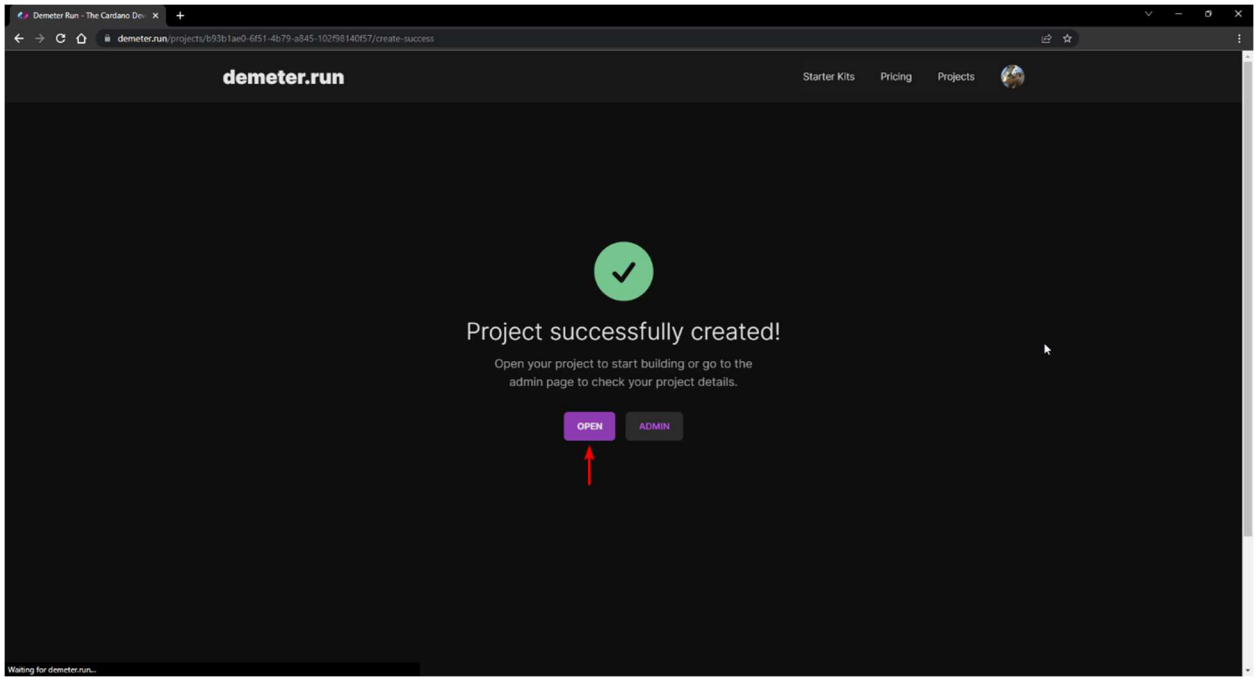
10. Next, you need to choose a network for your project. For testing purposes, we will typically use the **Preview** networks. Select the **Preview** network and click the **NEXT** button to continue, as illustrated below.



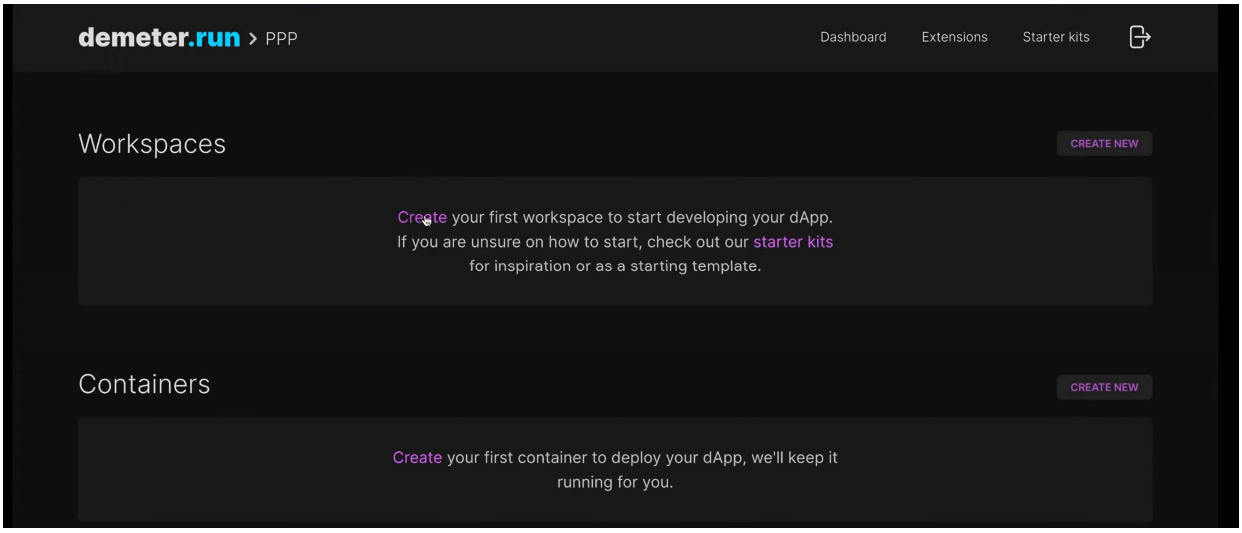
11. The last step is to provide a name for your project. Also, in this step, you can review the project's details; be aware that you can have only one project in the Discover plan. In this demo, the project was named **PPP Demo**. Set a name for your project and click on the **CREATE PROJECT** button to finish, as illustrated below.



12. Once your project is created, it is ready to start building applications. You will see a confirmation message, as illustrated below. Next, you can click the **OPEN** button to navigate to the project's Development Console.

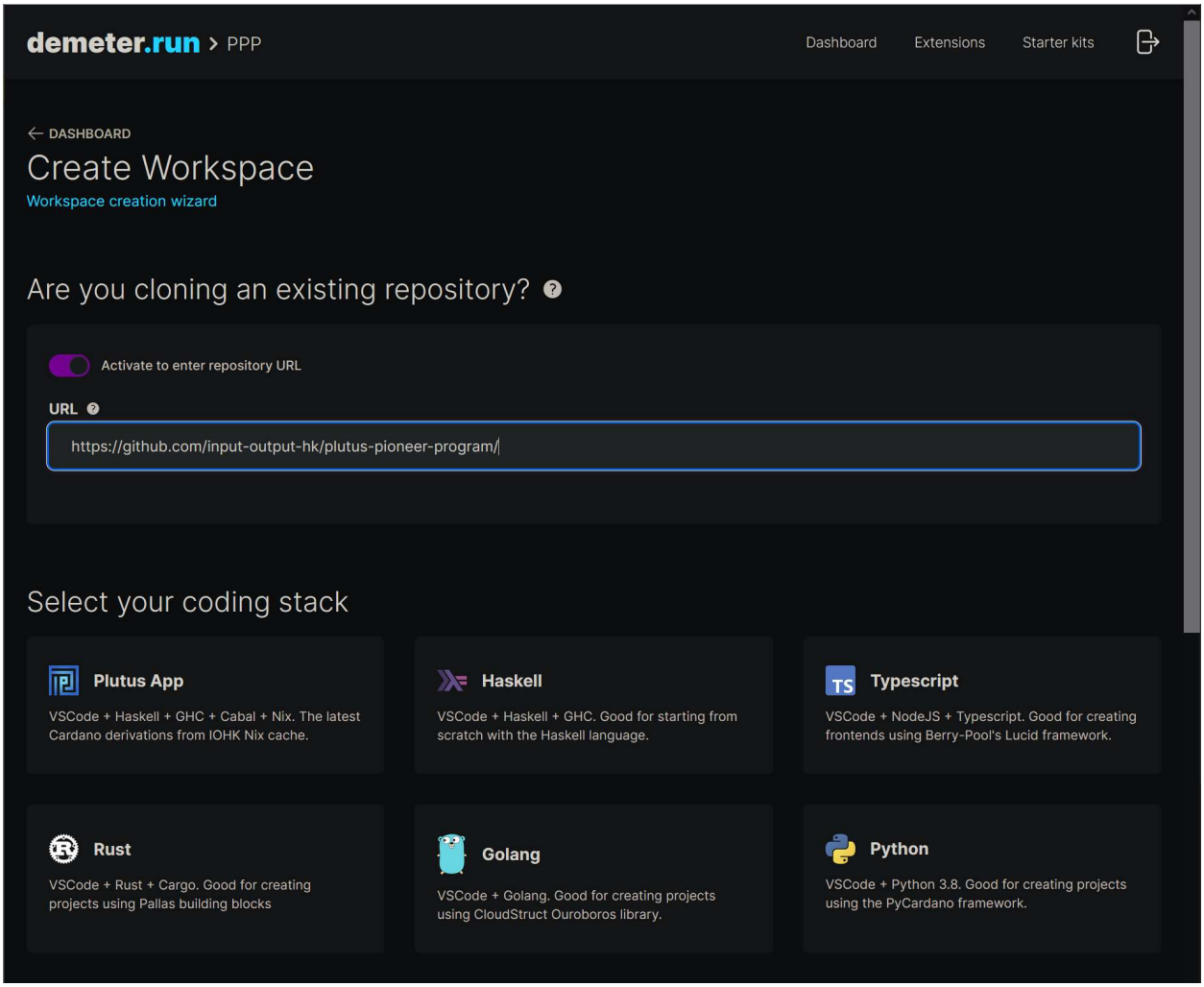


13. After that you see the dashboard. You will need to create a workspace that is tailored to your needs of development tools. Click **Create new** on the upper right corner of the Workspace section.

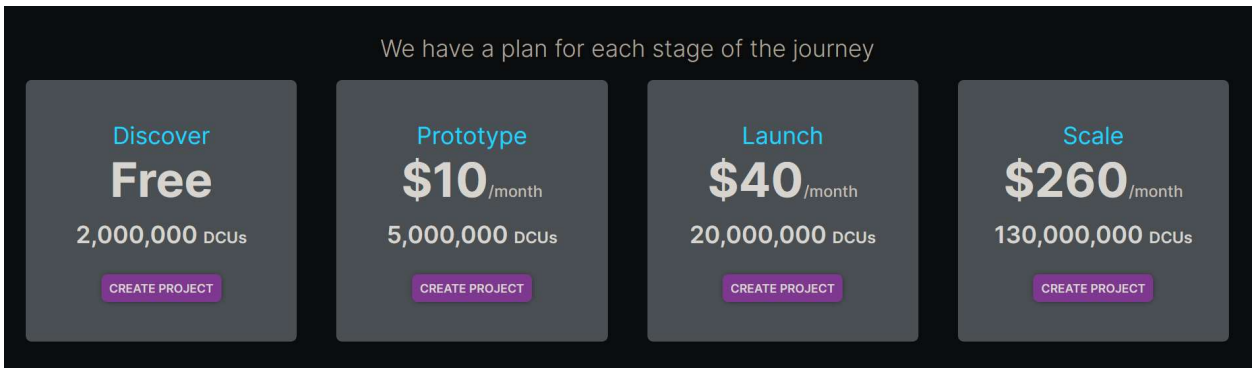


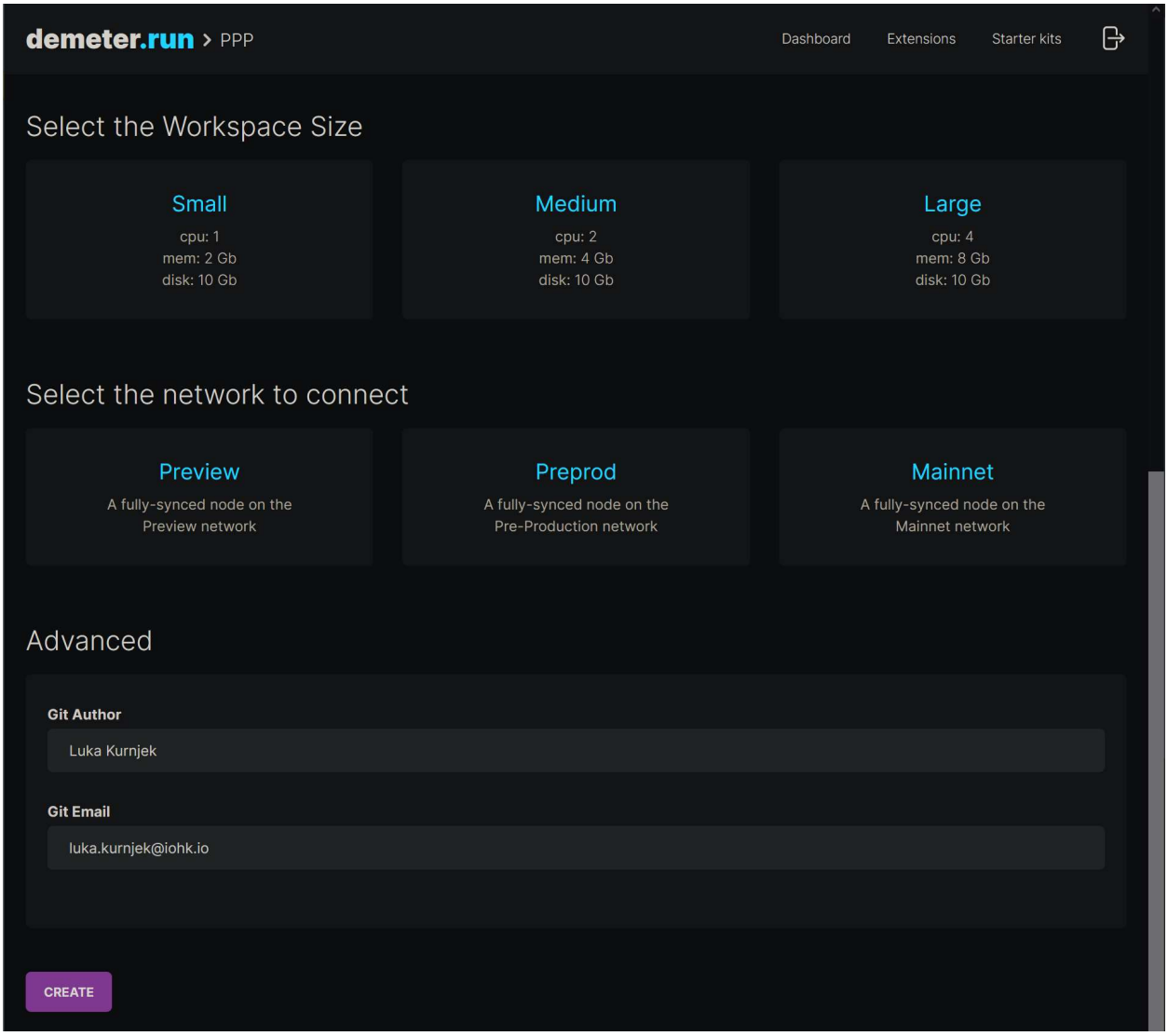
14. In the Create workspace section we click the toggle button for using an existing repository and input the Plutus Pioneer Program GitHub repository. If you have a GitHub

account, you can clone the PPP repo to your account and then input a link to your repository. In the **Select coding stack** section choose the Plutus App option.



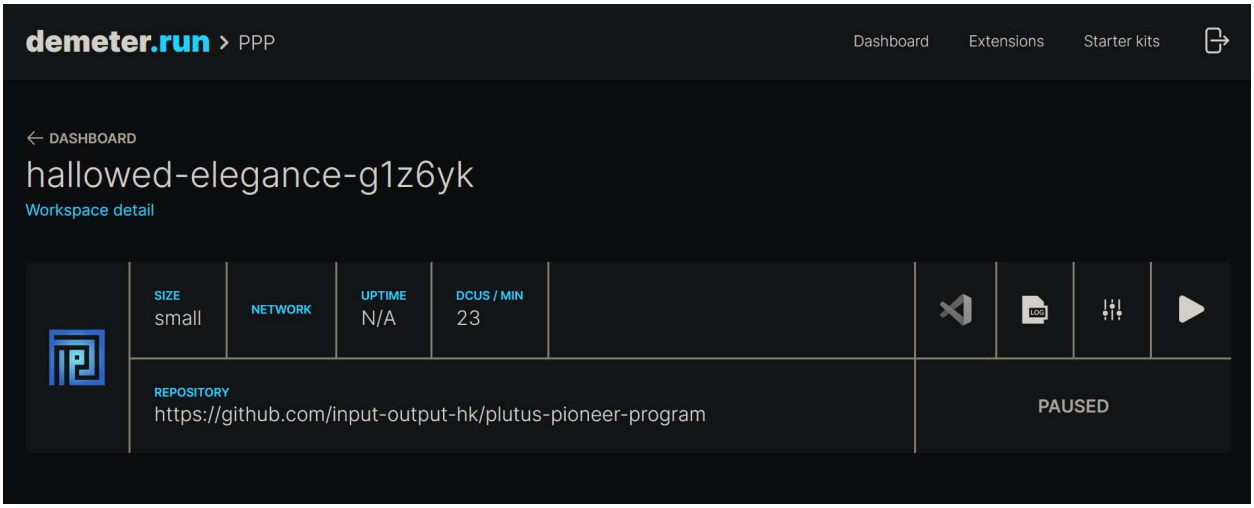
15. At the bottom of this page, you can select your workspace size. The bigger the size is the more credits it consumes. For our project we will choose medium size. You have 2.000.000 DCU (Demeter Computed Units) for free. For the network we select **Preview** and then click the **Create** button at the end of the page.



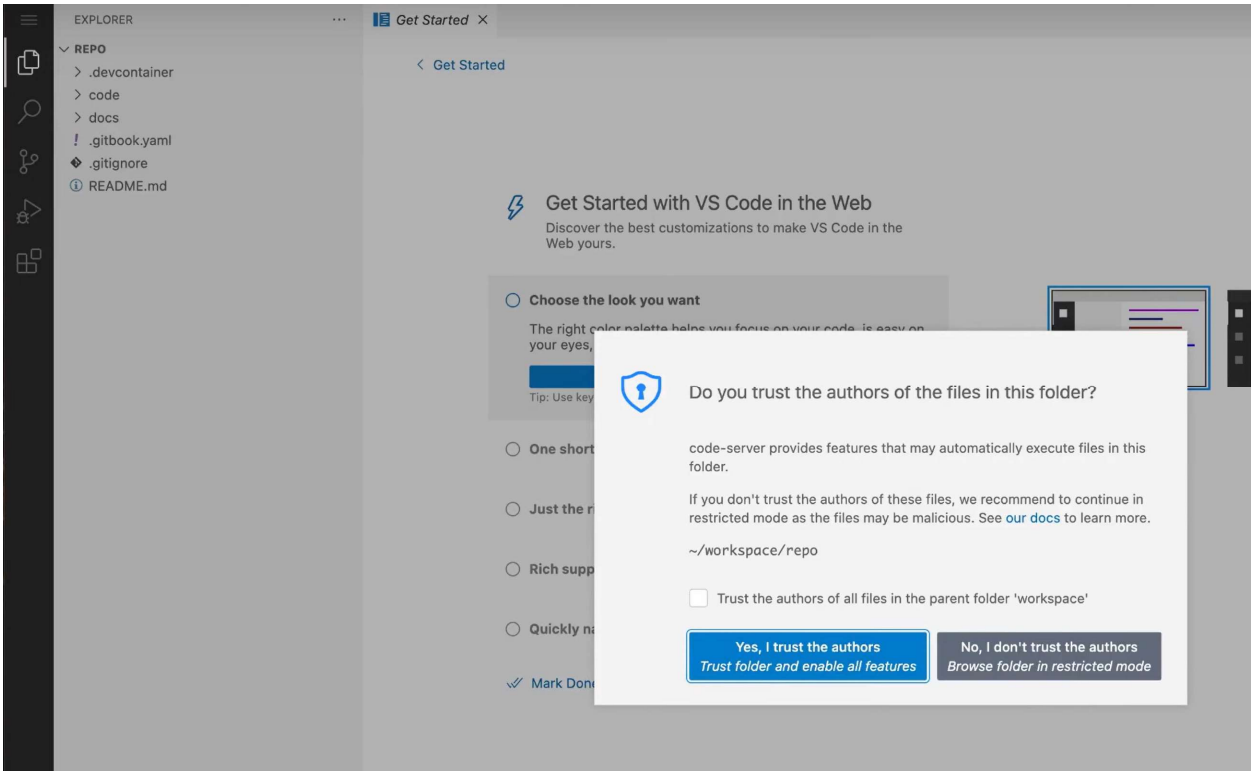


16. After that you will be redirected to the workspace detail page and the workspace information will be displayed. The workspace also needs some time before it builds. After its build, it will automatically go into run mode. You can click the **Pause workspace** button on the right side to pause it and then click again on the same button that will have the tool tip “Run workspace” to run it again.

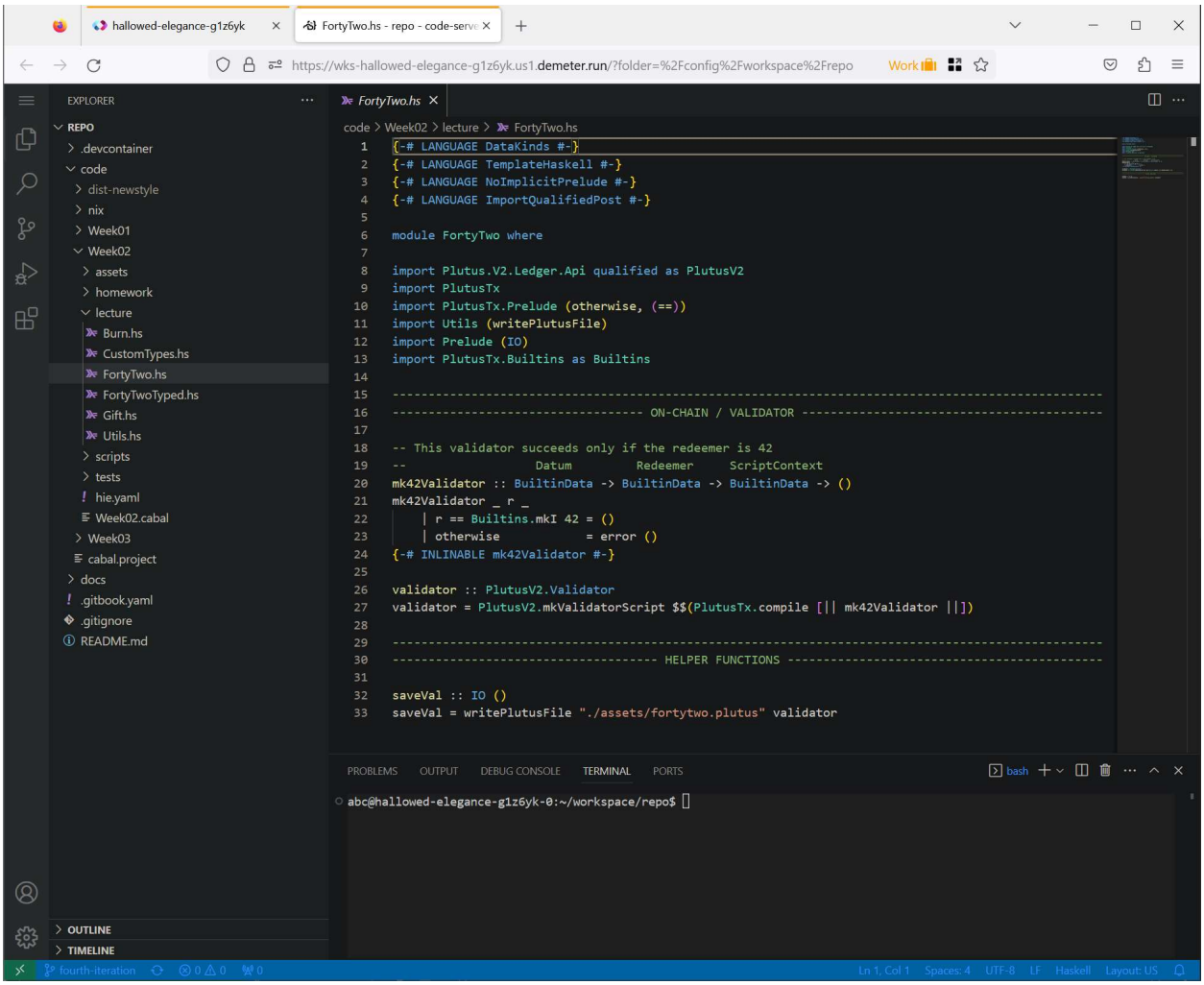
Demeter.run will charge you DCUs for a running workspace. So after you finish with your work, always pause your workspace.



17. Once everything is set up you can click the VSCode logo and a VSCode editor will appear that will contain all the files from the Pioneer Program repository you inputted in the workspace creation step. Check the trust box and click the **Yes** button.



18. Once you are inside the VSCode editor you can click on the menu button in the upper left corner and click **Terminal -> New Terminal**. This will open a terminal window in your workspace and certain command line tools will become available. The code for each lecture is contained in the *code/WeekXX/lecture* folders.



19. With the following commands you can update your project and create a validator script for the *FortyTwo.hs* code example from the Week02 folder.

```
$ cd code
$ cabal update
$ cd Week02
$ cabal repl
Prelude Gift> import FortyTwo
Prelude FortyTwo Gift> FortyTwo.saveVal
```

The file *fortytwo.plutus* is saved in the assets folder. You will learn in the next chapter what you can do with the code in the Week02 folder. If you have any questions, you can reach the PPP community on the IOG's technical community on Discord (https://discord.gg/inputoutput) by checking out the *#pioneers-questions* channel.

1.1.2 Installing Docker and VSCode

In this guide, you'll learn how to set up a local working environment using Docker and Visual Studio Code. If you have no previous experience using Docker and VSCode, you can follow the steps in this chapter that explain all necessary actions for a successful environment setup.

Use a host computer, not a virtual machine. The setup instructions will only work if you use Docker Desktop or Engine on your host computer. The setup inside a virtual machine will fail, whether it's a Windows or a Linux virtual machine.

Docker is a platform designed to help developers build, share, and run applications in an isolated environment on any operating system. To ease setting up a local working environment for this course, the IOG Education team created a Docker container that packages all the dependencies required to follow up the lessons of this program.

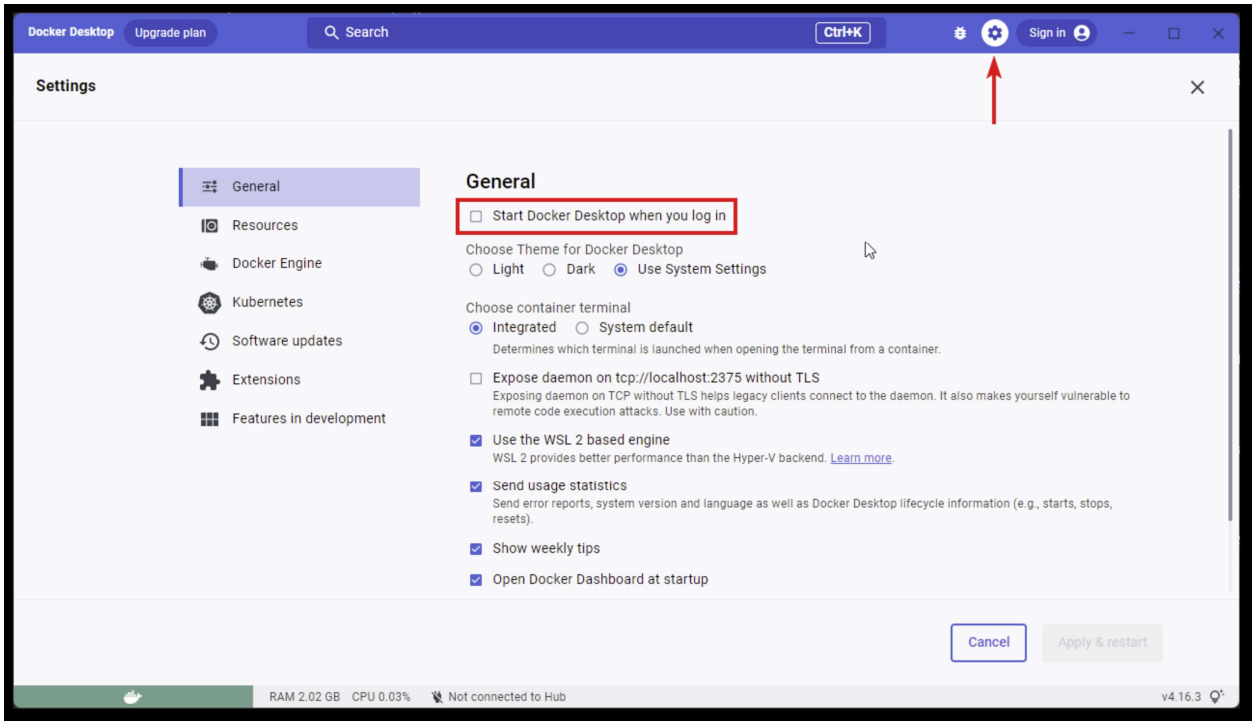
A Docker container is a standard unit of software that packages up code and all its dependencies, so an application runs quickly and reliably from one computing environment to another. From the Docker documentation, you can learn more about Docker containers on the following page: https://www.docker.com/resources/what-container/.

Follow the next steps to install Docker in your computer:

1. Open your browser and navigate to https://www.docker.com. Click the **Download Docker Desktop** button from the Docker's homepage. By default, you'll download a version compatible with your operating system. Docker is available for Linux, Microsoft Windows, and Apple macOS.

Important note for macOS users. Be sure that you download the correct version according to the chip of your computer, for M1 or M2 chips, download the "Apple Chip" version. For Intel chips, download the "Intel Chip" version.

2. After downloading the Docker Desktop installer, execute it and follow the instructions by choosing the default options. Installation options may vary depending on your chip and operating system. If you need detailed instructions, please visit the Get Docker section on the docker docs website: https://docs.docker.com/get-docker/.
3. When Docker Desktop is started, it automatically starts the docker daemon in the background. This will happen every time you login into your computer. You can change this behavior by turning off the **Start Docker Desktop when you log in** option in the Docker Settings configuration. Note that the docker daemon must be running when accessing the docker container in VSCode which we will explain next.



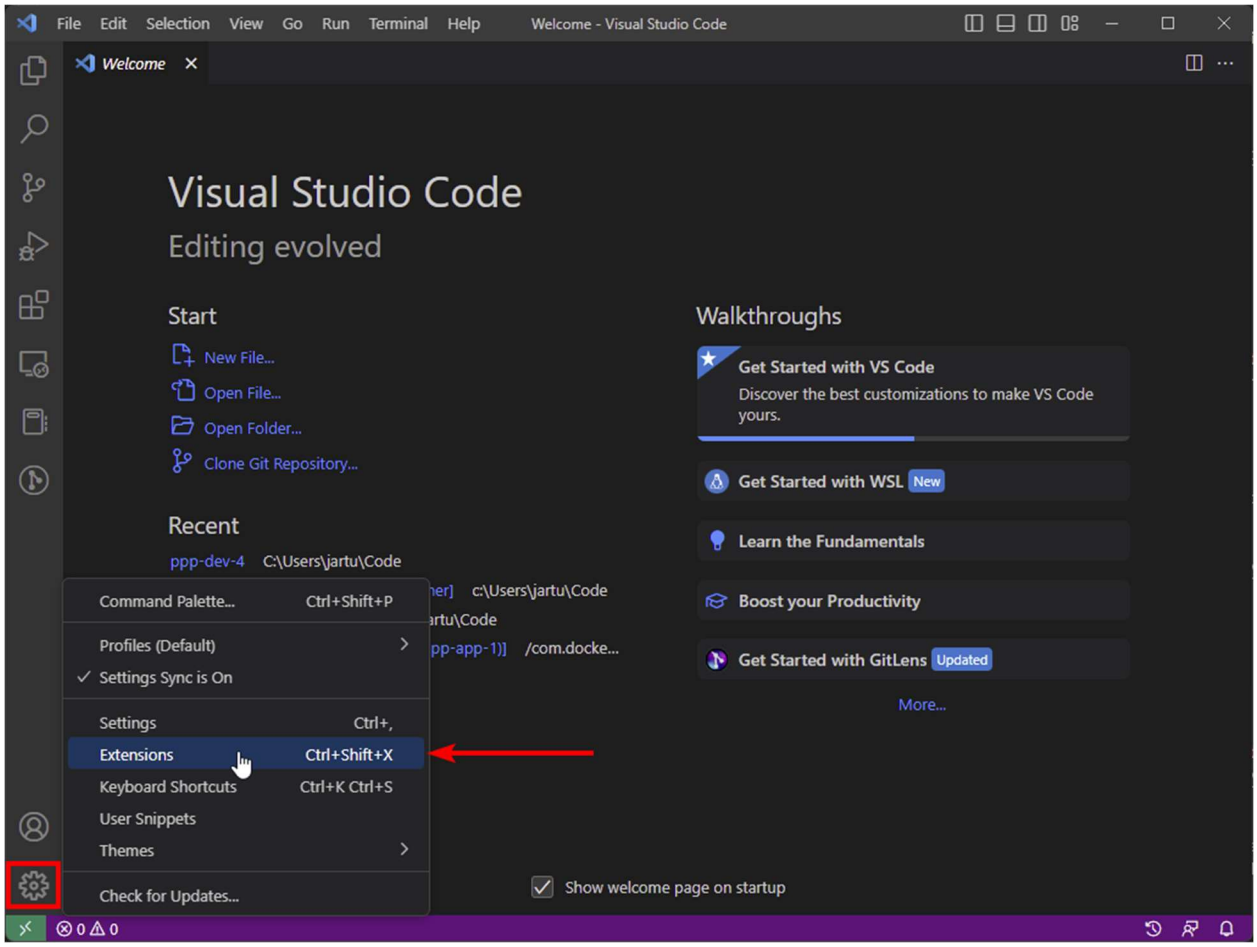
Visual Studio Code (https://code.visualstudio.com/), also known as VS Code, is a source code editor freely distributed by Microsoft that runs on Windows, Linux, and macOS. Additionally, to code editing, VS Code allows you to create and install extensions that eases your daily work.

Do not install any Haskell extension in Visual Studio Code. If you have VS Code installed, you may see several prompts from VS Code asking for permission to install several Haskell extensions when you open the PPP repository. You do not need to install any of those, as the Docker container we deliver will install all the Haskell extensions that VS Code needs. Some Pioneers who installed additional Haskell extensions report issues in completing this install guide; also, the IOG Education team members experienced problems while compiling Plutus scripts.

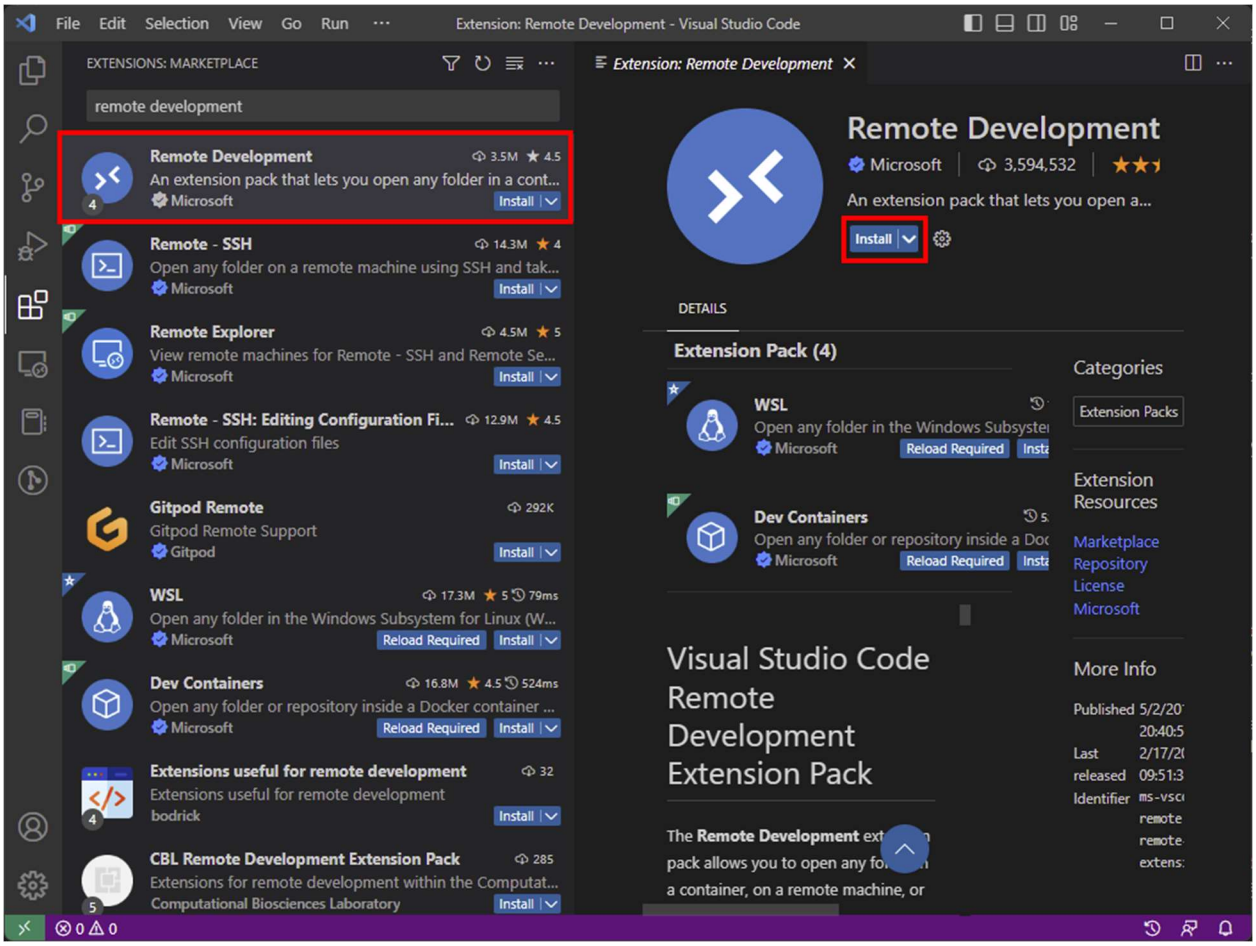
Follow the next steps to install VS Code and a handy extension that you will use in this course:

1. Open your browser and navigate to https://code.visualstudio.com/ to open the Visual Studio Code website. There is a download button that suggests you download the software depending on your operating system. Be sure to download the latest version.
2. After downloading the VS Code installer, execute it and follow the instructions by choosing the default options. Installation options may vary depending on your chip and operating system. If you need detailed instructions, please visit the Setting up Visual Studio Code section in the VS Code docs website: https://code.visualstudio.com/docs/setup/setup-overview

3. Once you have installed VS Code, open it to install an extension. Extensions are additional add-ons that extend the VS Code's functionality; Microsoft provides some extensions, but also, plenty of extensions are created by other companies and developers. To install an extension, click on the **Manage** icon in the bottom left corner and choose the **Extensions** options as illustrated below.



4. Next, a window showing the **Extensions Marketplace** will appear on the left side of the VS Code UI. In the search box, type **remote development** to look for the **Remote Development** extension provided by Microsoft: https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack
   Once the extension appears, click it. As illustrated below, you should click the **Install** button to install an extension.



After successfully installing the Remote Development extension, you will note that an **Uninstall** button appears in the extension description tab, and the **Open a Remote Window** green icon will appear on the bottom left corner of the VS Code UI.

You have now installed the required software to use the Docker container. Next, we'll guide you through the steps you must follow to load the Docker container and execute it for the first time.

1.1.3 Running the PPP Docker Container

Before moving forward into using the Docker container provided for this cohort of the PPP, you need to close VS Code. Please follow the next section, where we'll guide you through finishing the setup of your local working environment.

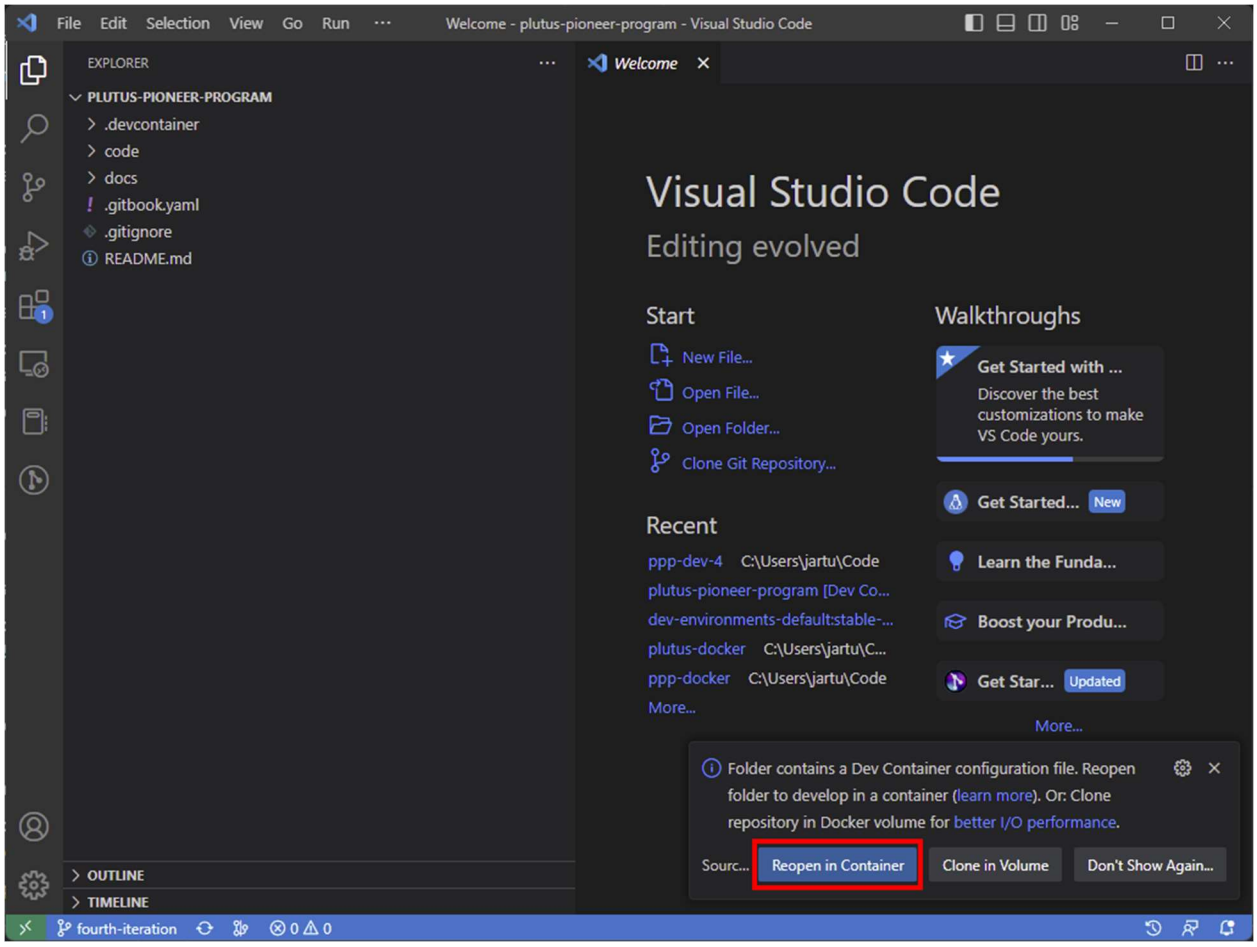
As stated in the demeter.run section you can work with the official GitHub repository or create a fork if you have a GitHub account and want to keep your work online. Instructions on how to create a GitHub fork can be found in the Plutus Pioneer GitBook at the following location:

https://iog-academy.gitbook.io/plutus-pioneers-program-fourth-cohort/preliminary-work/setup/docker#forking-the-ppp-repository

In either case you will need to clone a Plutus Pioneer repository to your computer using Git. If you don't have Git installed, please follow the instructions from the Git documentation https://git-scm.com/book/en/v2/Getting-Started-Installing-Git, where you'll find detailed information about installing Git on Linux, macOS or Windows. Once you have Git installed open up a terminal that has the git command available, cd into you desired location and clone the PPP GitHub repository:

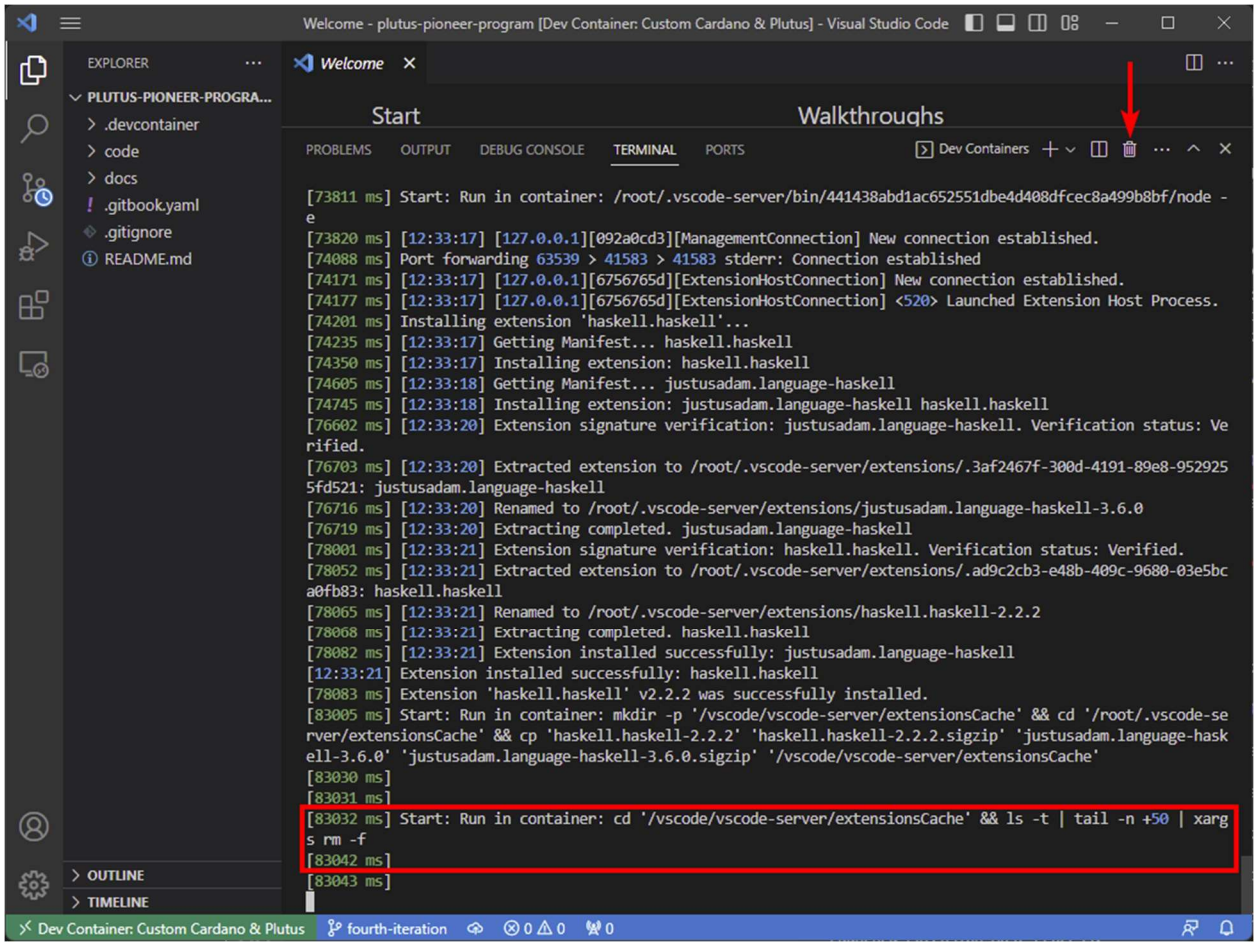
```
$ git clone https://github.com/input-output-hk/plutus-pioneer-program
```

Next open VSCode and in the File menu click **Open Folder** and then select the PPP folder you have cloned with Git. When VS Code opens, the **Remote Development** extension will detect the Docker container. As shown in the image below, you'll see a message asking to reopen your project in the container. Click the **Reopen in Container** button to continue.



After reopening your project in Container, the Docker container will be built. You can click the **Starting Dev Container** message to view the log. Please look at the log to be aware of when the container is ready. The build may take some time. After a few minutes, you will see a message

in the log starting with the text `Start: Run in container` as illustrated below. This message indicates that the dev container is ready.



You can safely close this terminal window by clicking the trash can icon in the upper right corner of the terminal window. Now, open a new terminal window into VS Code by clicking the **Terminal** menu and then **New Terminal**. From the new terminal window cd into the *code* and update the repository with the following command:

```
root@99286bb23f1c:/workspaces/plutus-pioneer-program# cd code
root@99286bb23f1c:/workspaces/plutus-pioneer-program/code# cabal update
```

This step is critical as all the code and updates should run into the code directory. Be patient while running these commands. Depending on your hardware configuration and an internet connection, the time required to execute the command `cabal update` may vary. It takes at least 5 minutes to finish; however, we experienced waiting times of up to 15 minutes in some hardware and internet settings. Now, to finish the dev container setup, type and execute the following command in the VS Code terminal to build all the dependencies required by Plutus.

```
root@99286bb23f1c:/workspaces/plutus-pioneer-program/code# cabal build all
```

After successfully running this command, you will see the system prompt back with no errors. Also, this command can take from 10 to 30 minutes to finish.

In case you are wondering what the docker container includes you can view the *Dockerfile* inside the *.devcontainer* folder. There you can also find the *devcontainer.json* file that specifies how the container is built. If the "image" field is uncommented it pulls the image from DockerHub. If the build field is uncommented the container is built from the local Dockerfile. Only one of them can be uncommented. If you add or update the commands in your Dockerfile you can then build a custom container for your needs. An example would be that you want to update the version of the Cardano node which is hardcoded in the Dockerfile to 1.35.5.

1.1.4 Troubleshooting Guide

The following issues may occur on any operating system:

- If you are getting this message from the Docker terminal in VS Code:

```
The connection to the terminal's pty host process is unresponsive, the terminals may stop working.
```

  The solution to this issue is to uninstall VS Code and install the Insiders version. You can download this version from this page: https://code.visualstudio.com/insiders/.

- **Linux Issues**

  If while opening the Docker container in VS Code, you get the following error message:

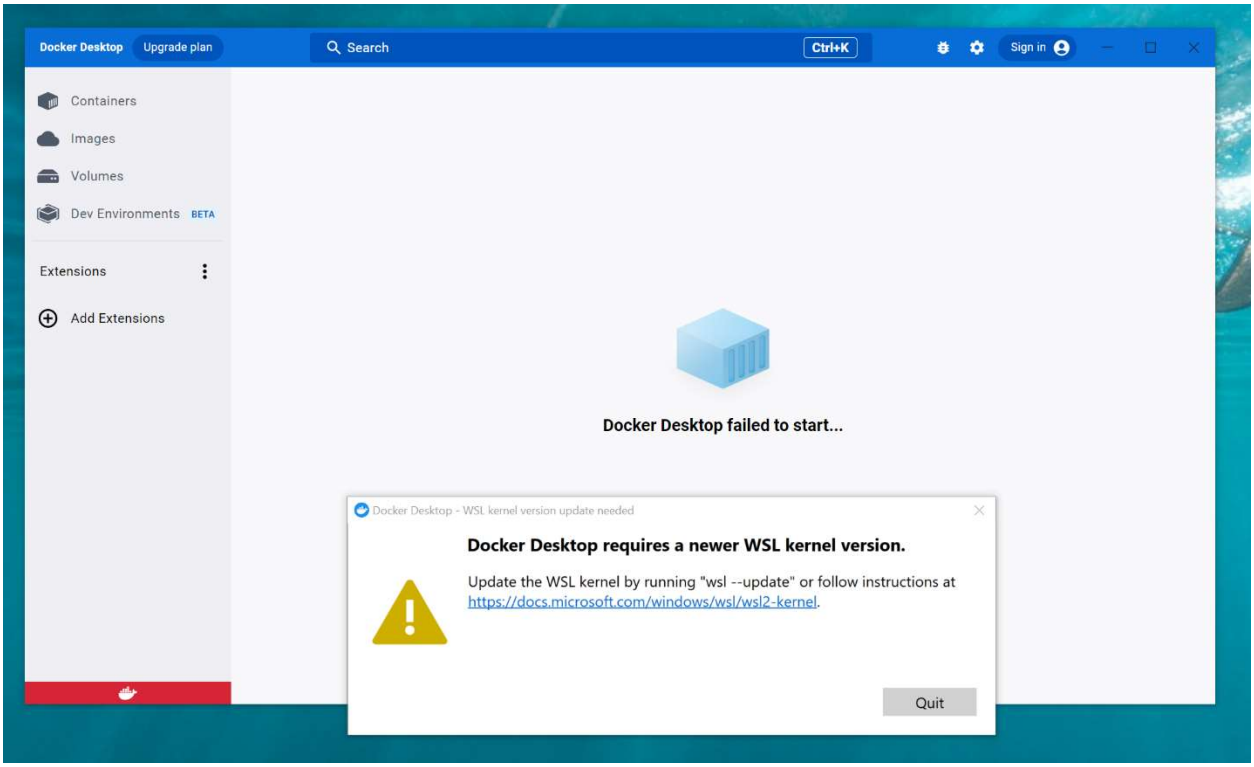
```
View container 'remote' requires 'enabledApiProposals: ["contribViewsRemote"]' to be added to 'Remote'.
```

  you should uninstall VS Code and install the Insiders version. You can download this version from this page: https://code.visualstudio.com/insiders/. To learn more about why this issue can happen, please read the Using Proposed API article https://code.visualstudio.com/api/advanced-topics/using-proposed-api from the Visual Studio Code documentation.

- **Windows Issues**

  If you have installed Docker Desktop before joining the PPP, you might receive an error message asking to install the latest WSL version. An example of the error message is shown in the image below. Please refer to this page from the Windows Subsystem for

Linux Documentation: https://docs.microsoft.com/windows/wsl/wsl2-kernel for further instructions.



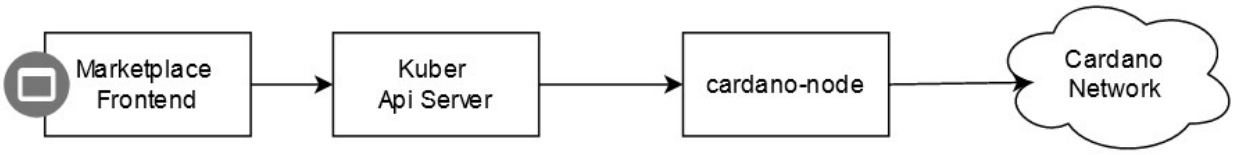
1.2 Kuber marketplace

In this chapter we will demonstrate a DApp that you can build yourself once you are done with this course. The DApp we are going to demonstrate is built by dQuadrant as an example to demonstrate the Kuber library. Kuber is an open-source Haskell library and rest API to easily compose complex EUTxO transactions on the Cardano blockchain. The DApp is called the *cardano-marketplace* and can be found at the dQuadrant GitHub page:

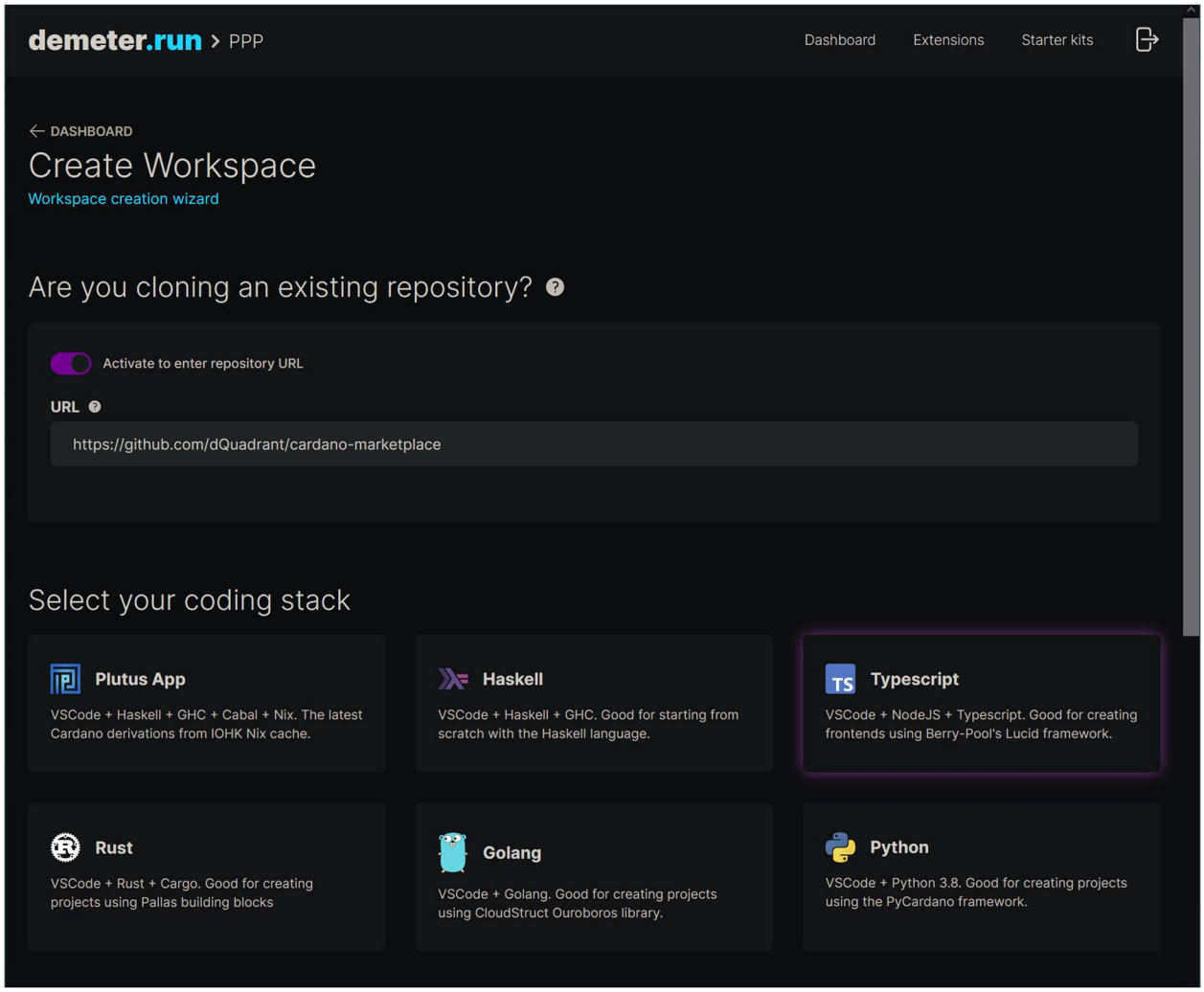
https://github.com/dQuadrant/cardano-marketplace

The marketplace can be deployed in multiple ways, and we're going to take the simple route and deploy a front-end for it, which does the following:

- connects to an existing Kuber server to construct transactions.
- lists tokens for sale using blockfrost API.



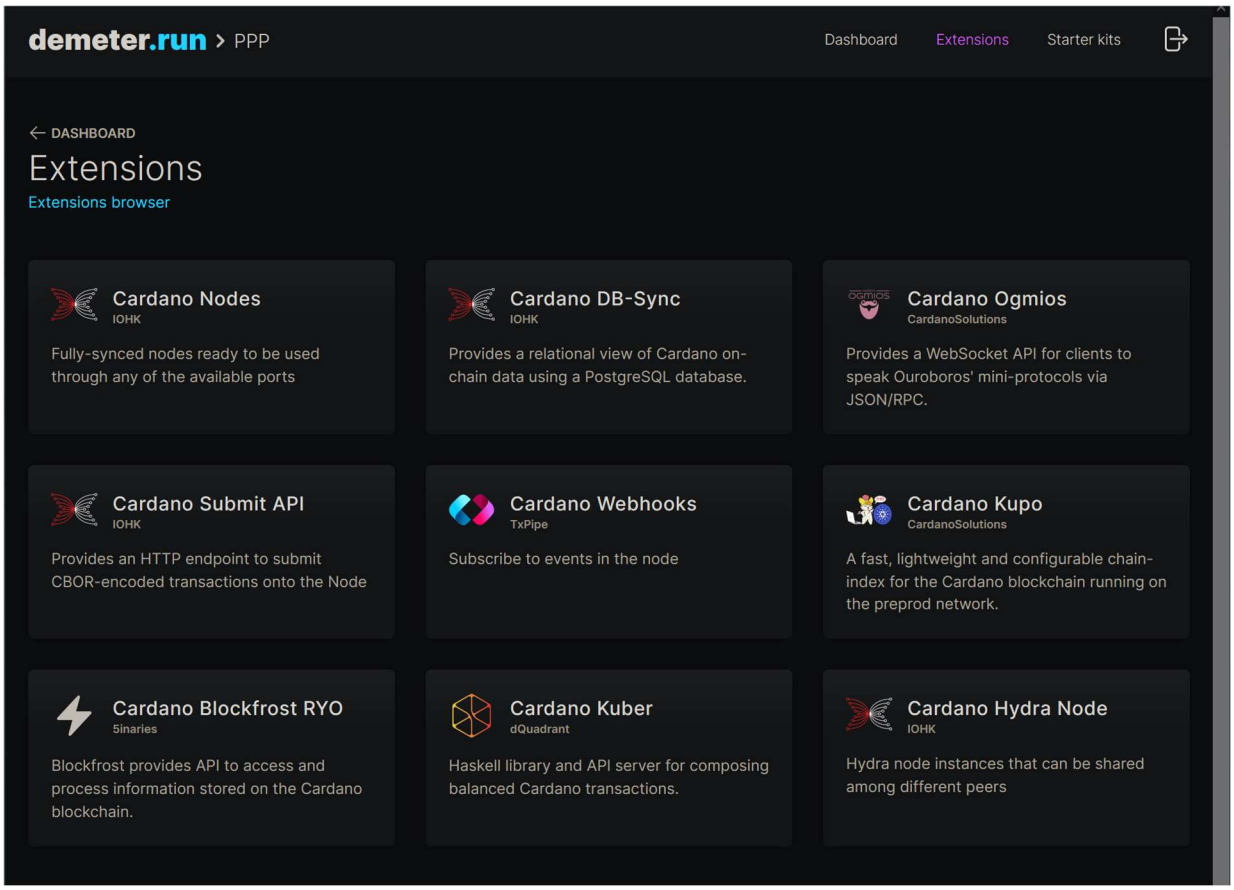
We are going to use Demeter to deploy this architecture. We go to demeter.run and open the project we already created in chapter 1.1.1 Using Demeter.run. If you have not performed these steps go to the chapter and follow the instructions up to creating a workspace. Then click **Create New** under the Workspace section. Input the GitHub link of the cardano-marketplace and select the Typescript coding stack.

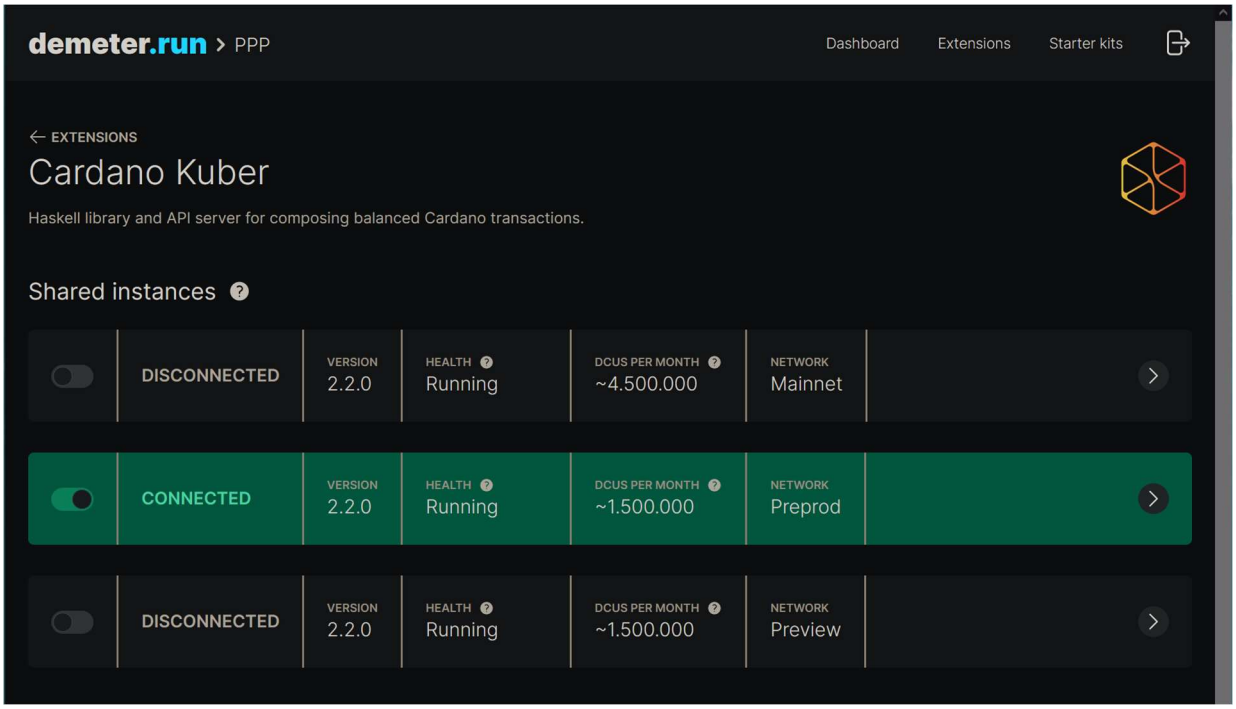


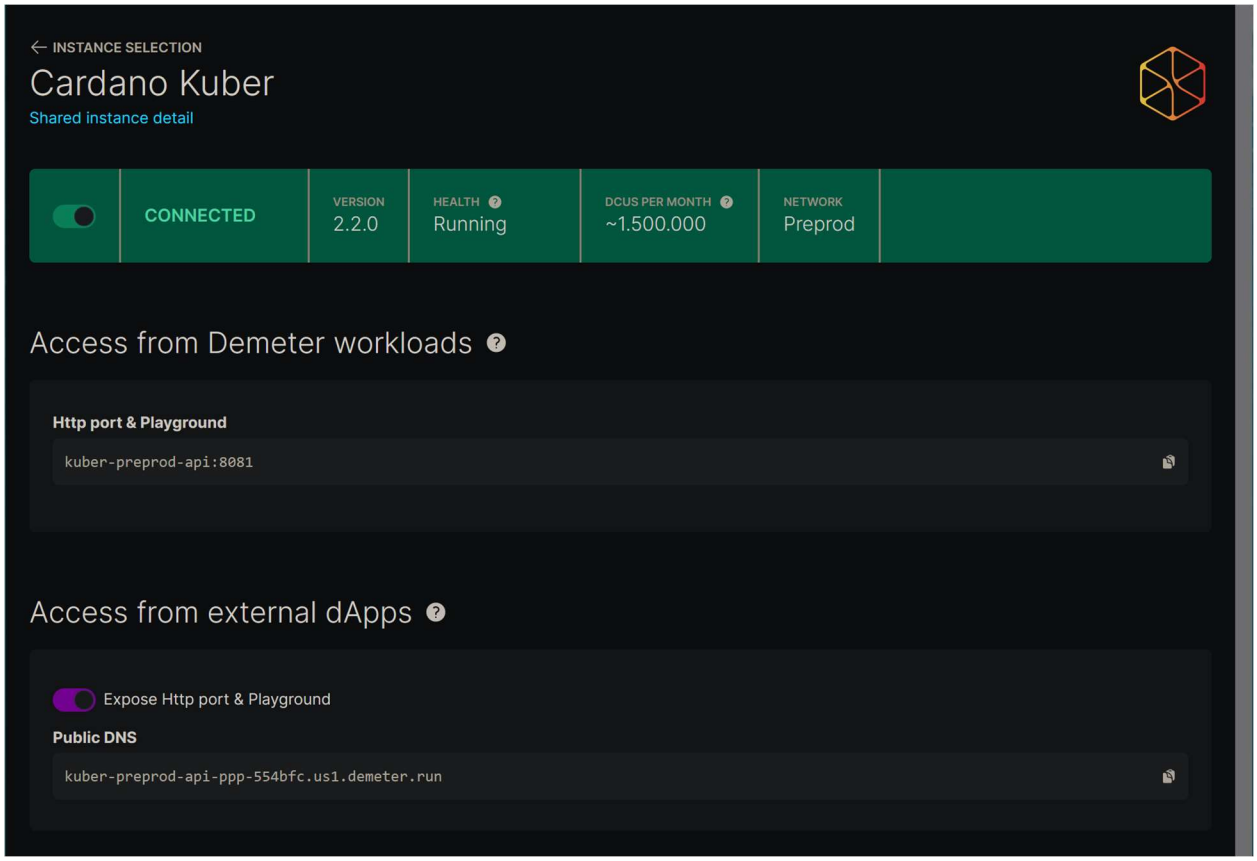
At the bottom of the page select the small workspace size and the Preprod network. Because we need a Kuber API to connect with the cardano network we will connect the Kuber extension to our workspace. We click the **Browse** extension in the Connected extensions section and choose Cardano Kuber. Next under the Shared instances section we turn on the Preprod net and click the arrow on the right side.

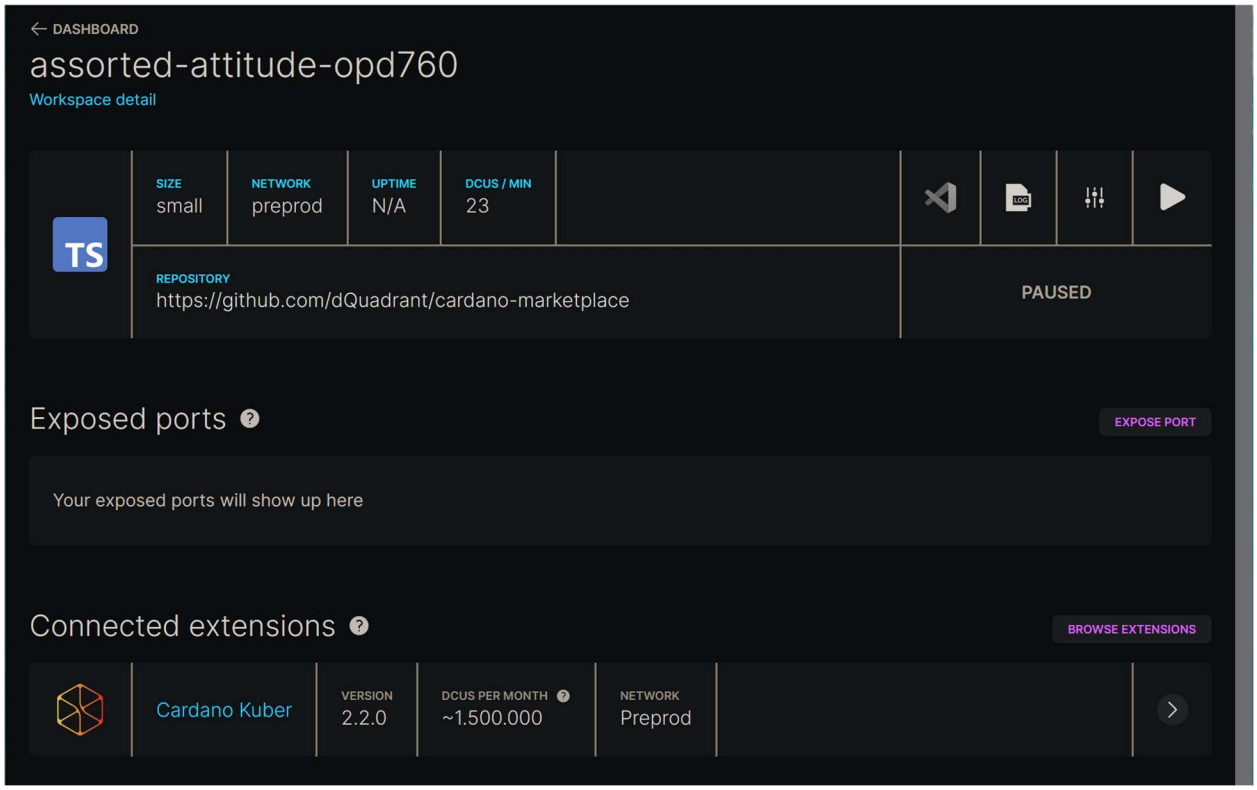
In the access from external dApps section we toggle the button Expose Http port & Playground, which will open an http port of our dApp application such that we can access it from our browser. We copy the URL under the Public DNS section that gets displayed. Then we return to

the dashboard by clicking on the top of the page **Instance selection -> Extensions -> Dashboard**. We can then enter our workspace by clicking the VSCode button.





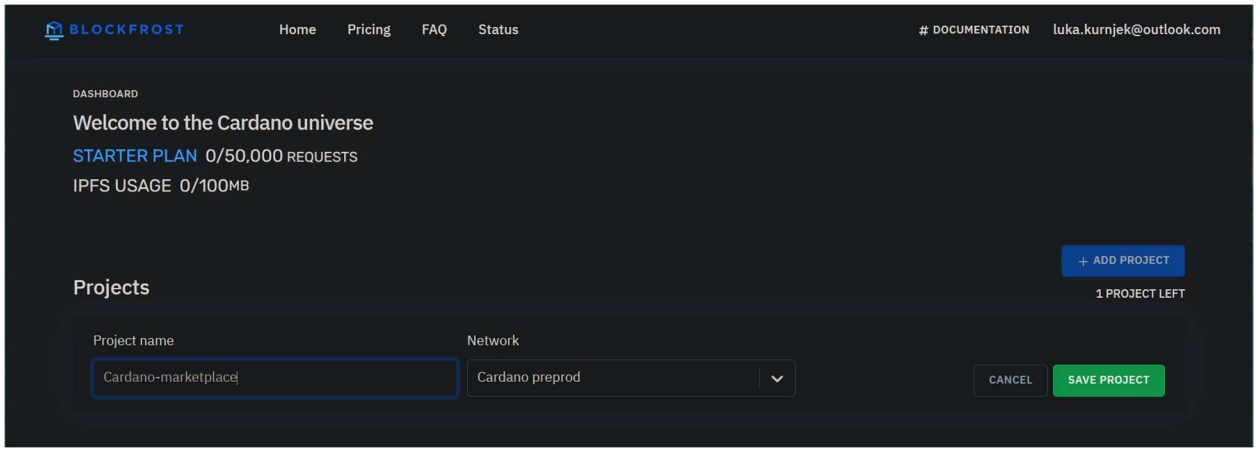

INSTANCE SELECTION
Cardano Kuber
Shared instance detail
CONNECTED
VERSION
2.2.0
HEALTH
Running
DCUS PER MONTH
~1.500.000
NETWORK
Preprod
Access from Demeter workloads
Http port & Playground
kuber-preprod-api:8081
Access from external dApps
Expose Http port & Playground
Public DNS
kuber-preprod-api-ppp-554bfc.us1.demeter.run


DASHBOARD
assorted-attitude-opd760
Workspace detail
SIZE
small
NETWORK
preprod
UPTIME
N/A
DCUS / MIN
23
REPOSITORY
https://github.com/dQuadrant/cardano-marketplace
PAUSED
Exposed ports
EXPOSE PORT
Your exposed ports will show up here
Connected extensions
BROWSE EXTENSIONS
Cardano Kuber
VERSION
2.2.0
DCUS PER MONTH
~1.500.000
NETWORK
Preprod

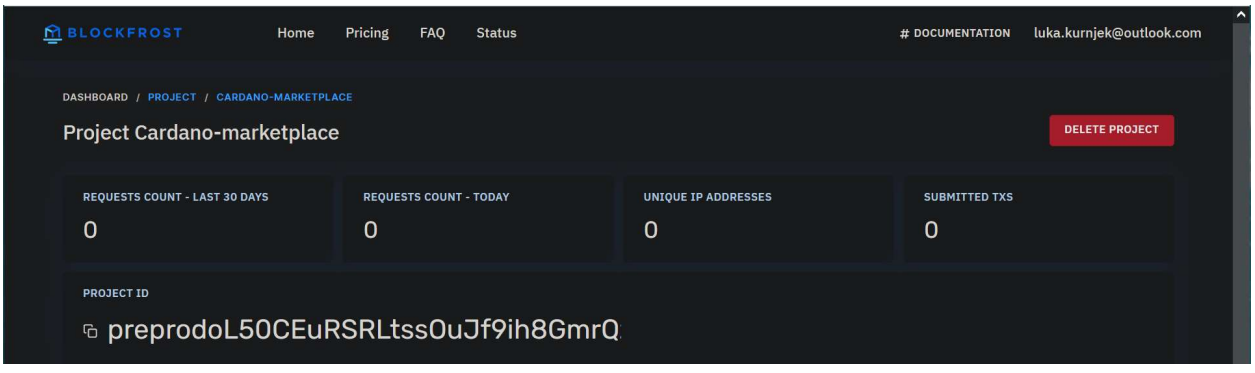
Before we build the frontend, we first need to configure it. Inside the VSCode editor we go to *frontend/src/config* and edit the file as follows:

```
export const kuberApiUrl = "https://kuber-preprod-api-ppp-554bfc.us1.demeter.run"
export const explorerUrl = "https://preprod.cexplorer.io"
export const blockfrost = {
    apiUrl: "https://cardano-preprod.blockfrost.io/api/v0",
    apiKey: "preprodoL50CEuRSRLtssOuJf9ih8GmrQzXgaiy",  // replace the api key
}
```

The link under the *kuberApiUrl* is the one we copied before when we exposed the http port. We also change the URLs to Cardano pre-production network. Under the API key we input the API that you can obtain from blockfrost.io. Go to their webpage and create an account or sign in with GitHub or Google account: https://blockfrost.io/auth/signin. After that under the projects section input your project name, select the Cardano preprod network and click **Save project**.



After that under the section project ID your blockfrost API key will be displayed.

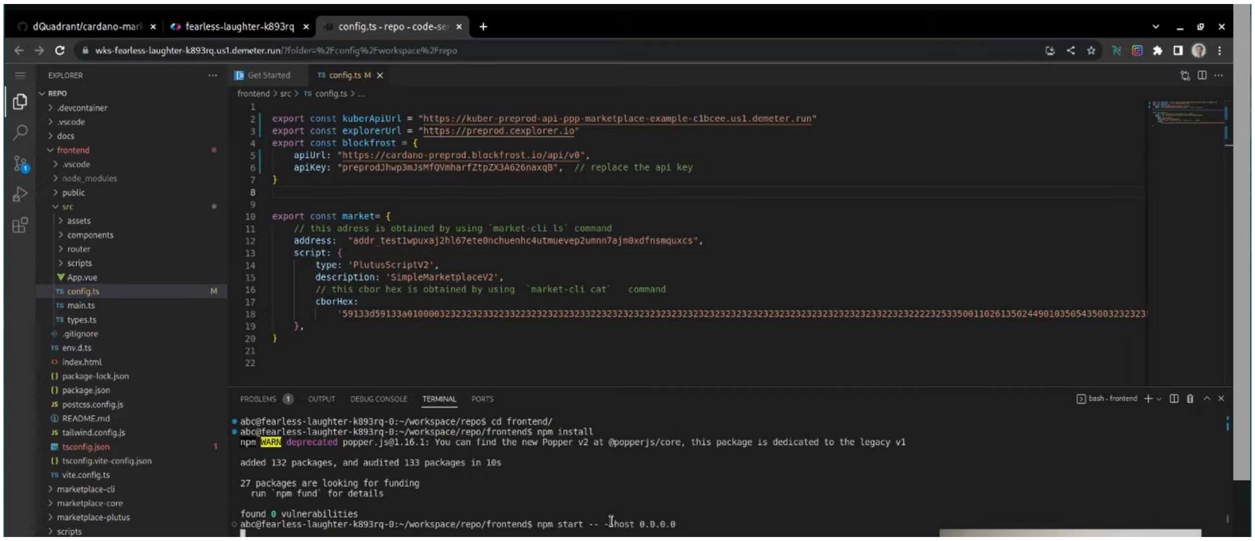


Then we got to the menu **Terminal -> Open Terminal** and in the open terminal we cd into the frontend directory and install the frontend with the npm command:

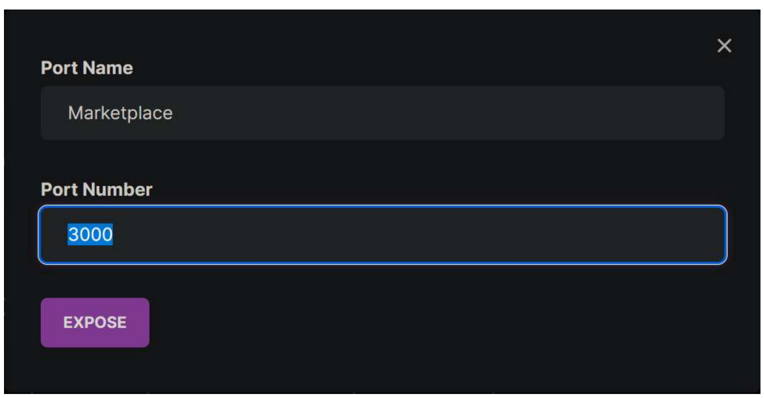
```
~workspace/repo/frontend> npm install
```

Once the installation is done, we are going to run the front-end by starting the web server:

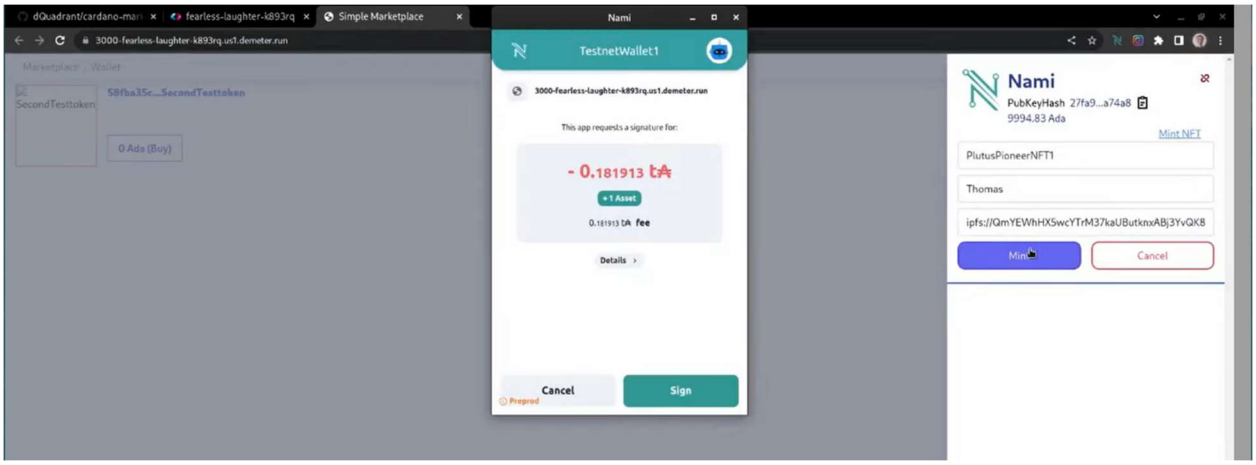
```
~workspace/repo/frontend> npm start -- --host 0.0.0.0
```



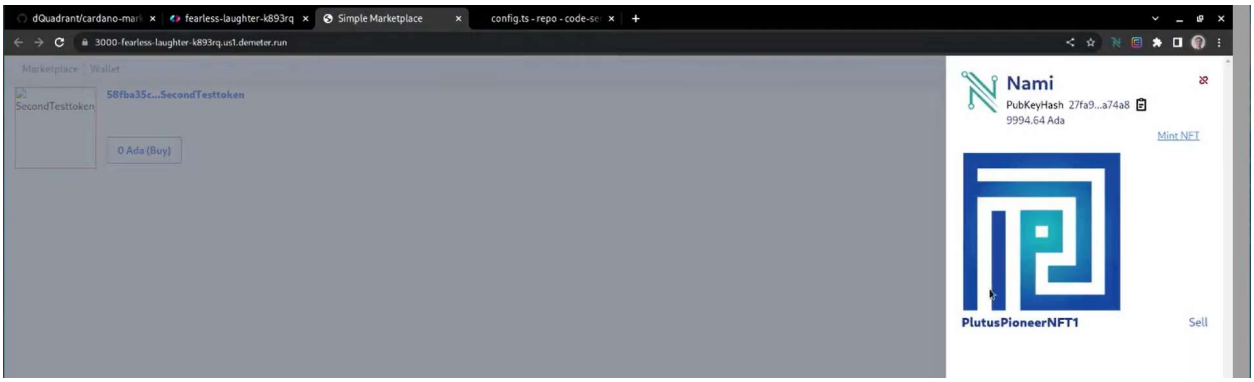
This will open a local host at port 3000 but since you are using demeter this is not reachable from your web browser. You need to go back to your settings on the workspace and in the section **Exposed ports** click on **Expose port**. Under name you can type in **Marketplace** and under port type in 3000 and click **Expose**.



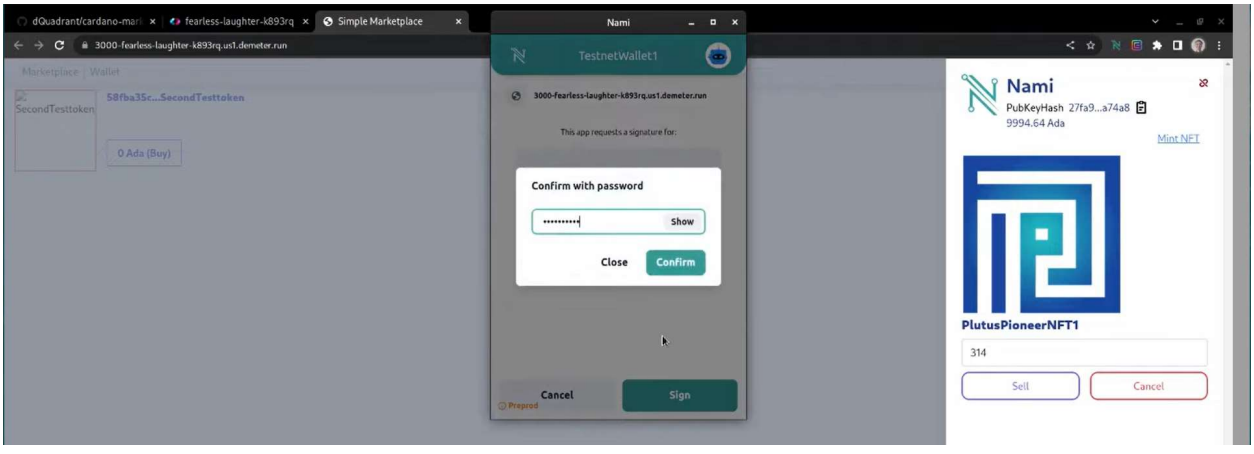
Once this is done you will see a link that you can use to access your marketplace webpage that you have created. You will see on the marketplace one token that is listed for sale. In the upper tab of your marketplace webpage, you can also choose to connect a wallet that you have installed in your web browser. Examples of such wallets are Nami and Eternl that have to be connected to the preproduction network. From the Nami wallet you can click on the MintNFT button and input your token name, artist name and image URL. It will prompt you to sign a transaction. You will of-course need some test ADA in your wallet address to mind the NFT.



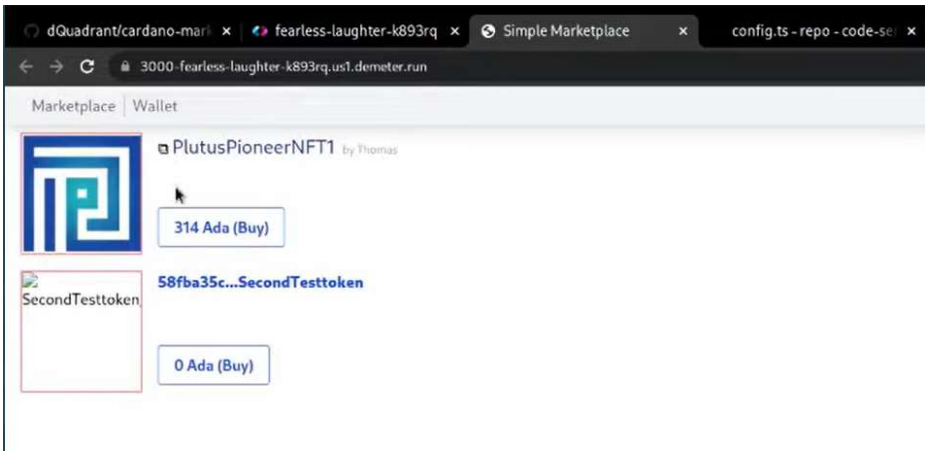
To receive test ADA you can go to the Cardano Testnet faucet and request some ADA for the Preprod testnet: https://docs.cardano.org/cardano-testnet/tools/faucet. After you minted your NFT you want to sell it and connect to the Nami wallet again and click on your NFT that will be shown in your wallet. There will be a Sell highlighted text that you can click.



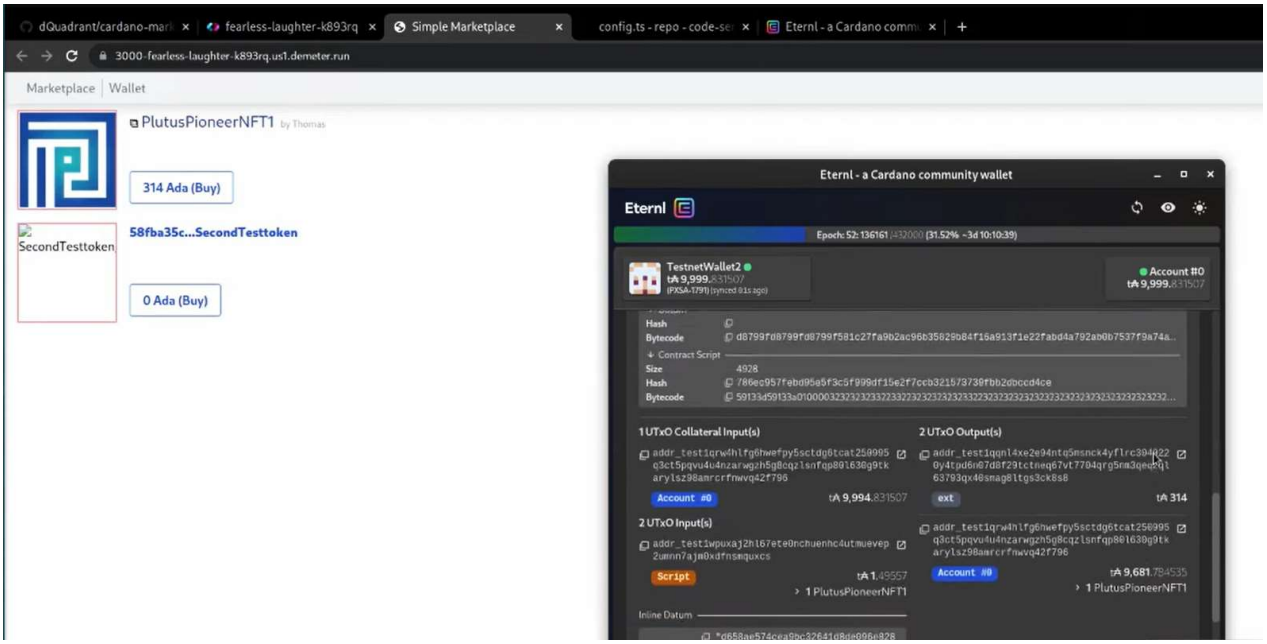
You can then input for how much ADA you want to sell it and click on the **Sell** button and sign again the transaction that will put the NFT on the marketplace.



You can now see the NFT on your marketplace and also the price that you have set is visible.



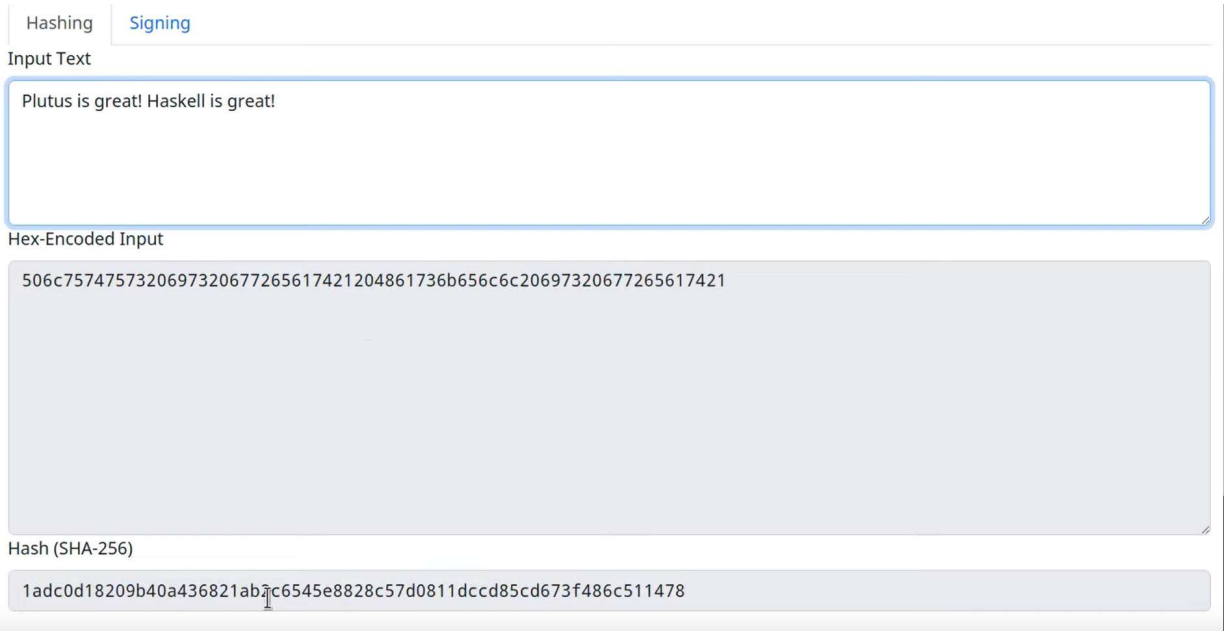
If you want, you can now buy this NFT with another wallet that should be also connected to the pre-production network. You simply have to click the **Buy** button and choose your wallet. If we do this with the Eternl wallet a window will pop-up and display the transaction details.



We sign the transaction, and after it is processed we can see that we get the NFT in our Eternl wallet and the ADA amount for which we were selling the NFT will be sent to our Nami wallet.

1.3 Hashing and digital signatures

These are two cryptographic concepts that play an important role in Cardano and blockchain technology. Hashing takes as input a byte-string. For that we usually use a hexadecimal text that we get when we convert normal text into hexadecimal format. A hashing algorithm takes this hex byte-string and produces another byte-string as output as illustrated below.



Note in the image above the output byte-string always has the same length no matter the input length. If we use the SHA-256 hashing algorithm the output will always be 256-bits (32-bytes) long. Because one byte can be represented with 2 hexadecimal characters, the SHA-256 hash is 64 characters long. Hexadecimal characters can be numbers from 0 to 9 or letters from A to F.

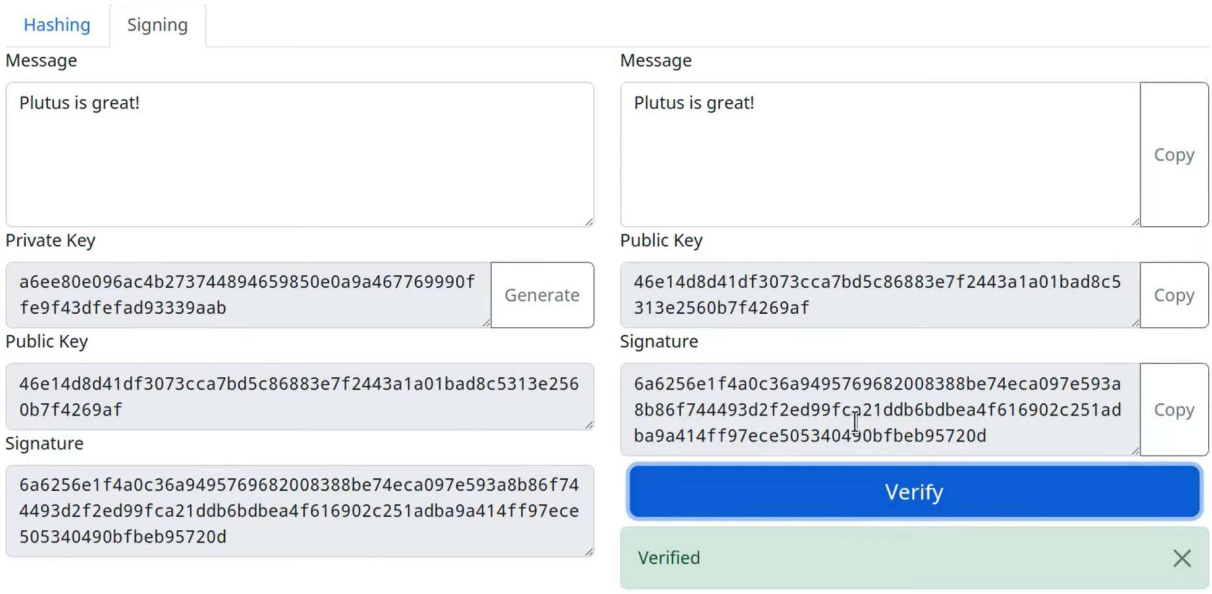
The second thing to mention is that hashing is very unpredictable. If you make a small change to a long text message the hash of the message will change entirely. Hashing algorithms are designed in a way that makes it practically impossible to reverse engineer the input from a hash value. That is a property on which Cardano and other blockchain technology critically rely.

There is an infinite number of input text, but only a finite number of hashes, since their length is fixed which is 2 to the power of 256 for the SHA-256 algorithm. Because of this it is in theory possible to find two different inputs that would produce the same hash values. Such a hash collision has not been found for SHA-256 in practice yet.

Now let us talk about digital signatures. Their point is that you can for instance digitally sign documents. In the background the content is again transformed to a byte-string. Let's take for example a short message instead of an entire document, that is just a long collection of text. To sign a message, we need a key pair consisting of a private key that only the signer knows and a public key that also the verifier knows. The private key can be generated by a program and the public key is then generated also by a program taking the private key into account.

As there are many hashing algorithms there are many digital signing algorithms. The idea is now that after signing the message with our private key we get an output string that is similar to a hash. We call it the signature of the document we signed. Another person who gets our

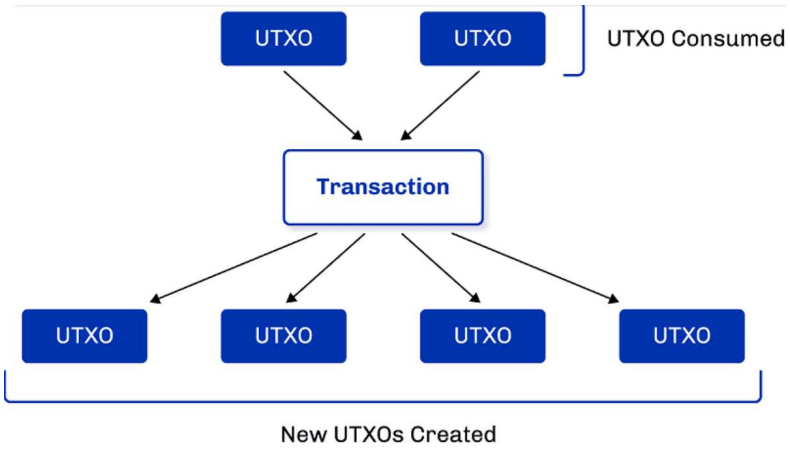
message, our public key and the signature can check with a program that the signature of the message he received belongs to the public key. Public keys are publicly known. Someone sending a signed email can add at the bottom of the email below his name, also his public key.



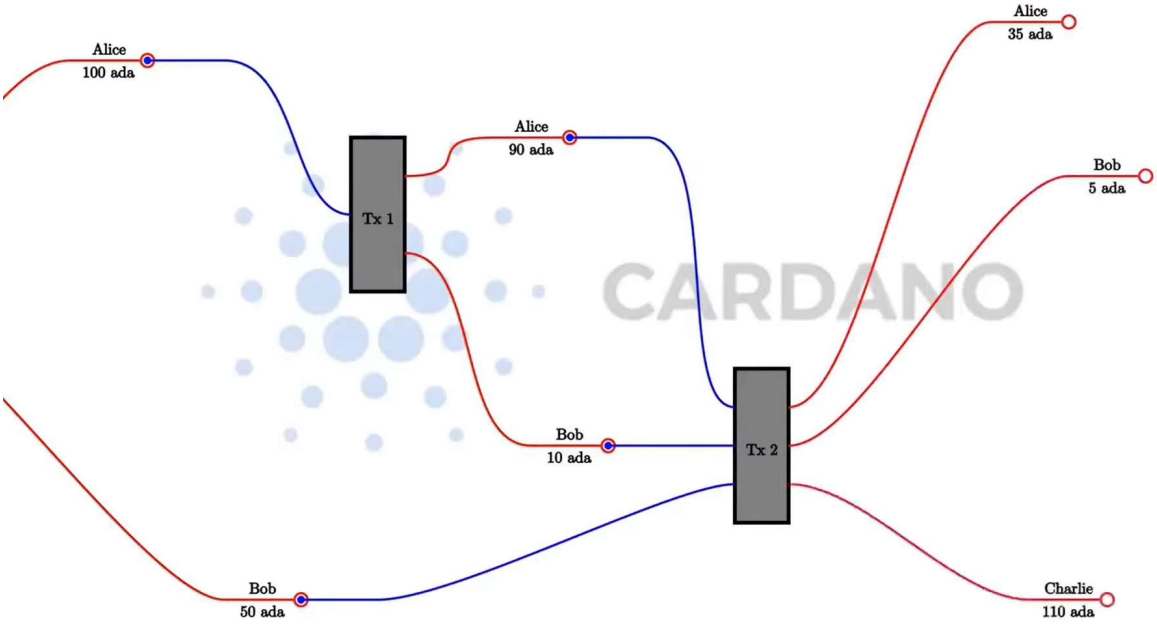
A thing to point out is that a signature is valid only for a specific message and the specific private key. If you change any of them then the verification will fail. Also, this technology is used in Cardano and other blockchains. It is digital signatures that keep your money safe. For a transaction to spend your money you must digitally sign it with your private key. And your public key in terms of transactions is your address. Public addresses and the funds they hold can be looked up by querying information that is stored in the blocks of a blockchain. The thing that is harder to figure out is which public address is associated with which person on the blockchain. If you don't publish your address together with your name people, then in broader terms you could say that you stay anonymous.

1.4 The EUTxO model

The Cardano blockchain uses the extended UTXO model (EUTXO) that is a variant of the Unspent Transaction Output (UTXO) model used by Bitcoin. Transactions consume unspent outputs (UTXOs) from previous transactions and produce new outputs, which can be used as inputs to later transactions. Unspent outputs are the liquid funds on the blockchain. Users do not have individual accounts, but rather have a software wallet on a smartphone or PC which manages UTXOs on the blockchain. It can initiate transactions involving UTXOs owned by the user. Every core node on the blockchain maintains a record of all the currently unspent outputs, the UTXO set. When outputs are spent, they are removed from the UTXO set.



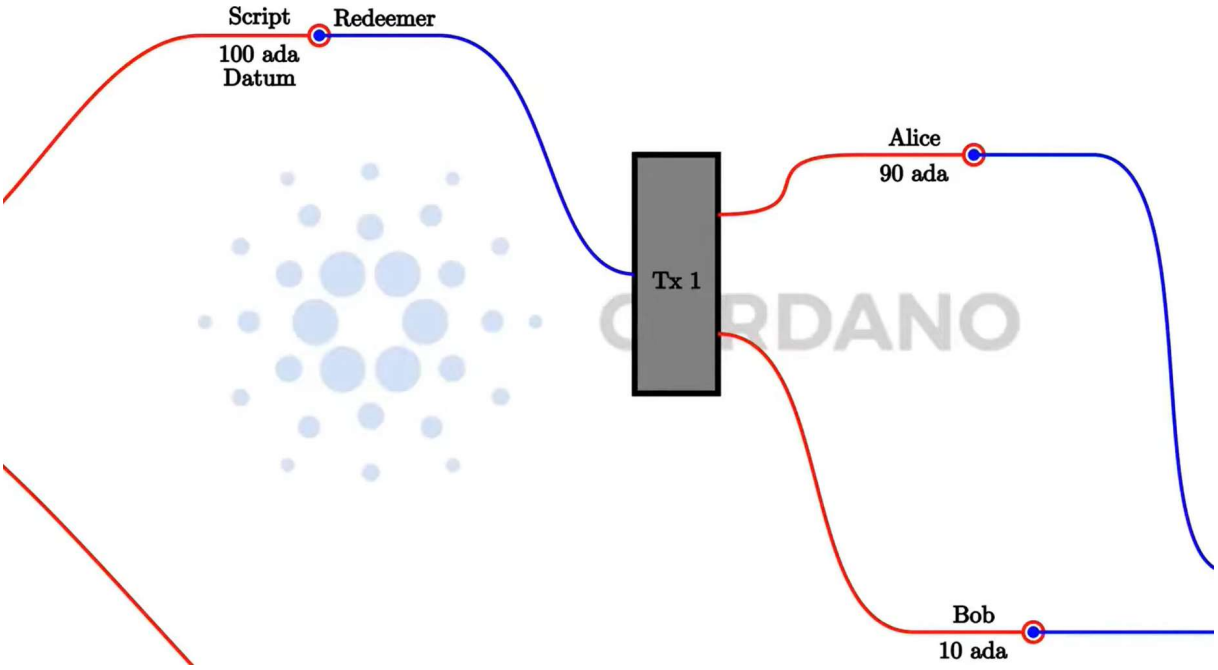
There are models other than UTXO. Ethereum, for example, uses an account-based model, which is what a normal bank uses. There everybody has an account, and each account has a balance. If you transfer money from one account to another, then the balances get updated accordingly. But in the UTXO model, the input is always the entire balance of an UTXO, and the outputs are newly created UTXOs from which one of them could belong to the user that provided his UTXO as input and would represent his change amount. Let's say Alice has 100 ADA and wants to send Bob 10 ADA. After that she wants to send to Charlie 55 ADA and in the same transaction also Bob wants to send Charlie 55 ADA.



In the picture above you can see that the inputs are always entire UTXOs, and outputs are newly created UTXOs that can belong to different participants, depending on the transaction.

As soon as an output is used as input in a transaction, it becomes spent and can never be used again. The UTXO output is associated with an address which is represented by a public key hash. We call them public key addresses. The ADA amount and optionally native tokens of a public key address is the sum of ADA and native tokens from all UTXOs belonging to this address. A transaction must be signed by the owner of the private key corresponding to the address that defines the input UTXO. Think of an address as a 'lock' that can only be 'unlocked' by the right 'key' – the correct signature. The user which controls a private key of an address can create transactions and use the ADA or native tokens sitting at the UTXOs of this address. For every transaction there is also a fee to pay denominated in ADA for the Cardano blockchain.

The extended UTXO model introduces in addition to public key addresses also script addresses that can contain some logic. That logic defines under which conditions the UTXOs sitting at this address can be spent. The address is unlocked by a piece of data called the *redeemer*, which in the conventional UTXO model would be a private key. A UTXO also contains some data called the *datum*, beside the amount of ADA sitting at the address. The datum together with the redeemer and the transaction context are the input information for a script logic that then chooses weather this transaction is valid and can be processed by a node on the network.



You can check the validity of a transaction in your wallet. If it is valid, you can be sure it will be processed on the network, given the condition that all the UTXO inputs are still present at processing time. If they are not the transaction will simply fail and no fee will be charged to the user that sent the transaction. We call the script that validates a transaction the validator. The

script address is defined as a hash of the validator code written in Plutus core language. The script addresses are publicly known. We will talk about Plutus core in the next chapter.

As said the validator script takes the datum, the redeemer and the transaction context as input information. The input for the datum is collected from each UTXO individually that is sitting at a script address. That means if there are multiple UTXOs specified to be consumed in the transaction, the validation logic is checked for each of them separately. So, in each validation there is only one datum as input coming from one UTXO.

This limited view of the validator script that can see only inputs, outputs and the transaction context that will be processed, has a security advantage compared to the Ethereum model, where the script can see the whole state of the blockchain. That enables Ethereum's scripts to be much more powerful but for this reason it's also very difficult to predict what a given script will do. That opens the door to all sorts of security issues. It can be mathematically proven that every logic you can express in Ethereum you can also express in the extended UTXO model. And that makes it a much safer and reliable transaction model compared to Ethereum.

A transaction in the EUTXO model can be classified as a producing transaction that produces a script UTXOs or as a spending transaction that spends a script UTXOs. In general, every transaction except the genesis transaction takes at least one UTXO as input and produces at least one UTXO as output. So, we use the terms spending and producing only when we talk about script addresses. If we first send ADA to a script address, we call it a producing transaction. After that if we try to collect that ADA from the script address, we call this transaction a spending transaction.

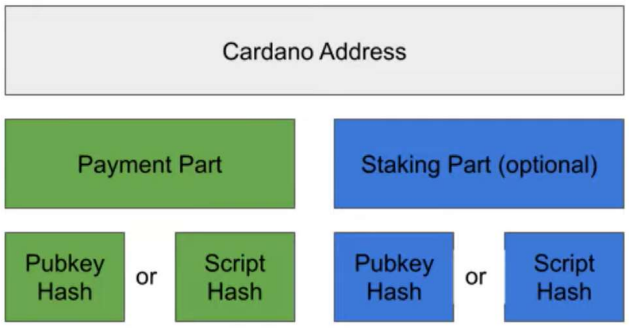
The producing transaction must include this address, and it must include the datum or the hash of the datum that will be attached to the UTXO created at script address. If it includes the hash of the datum, only a person that knows the datum by some other means not by looking at the blockchain is able to ever spend such an UTXO. The spending transaction is responsible for providing the redeemer, the transaction context and optionally also the datum. It also must provide the validator script. If we construct a transaction where the funds go to a public key address only a signature with the private key of the sending address is needed.

Let’s look now in more detail how the datum or its hash gets provided by the producing transaction. There are three options to do this. The first option is that the output contains only the hash of the datum. Then the consuming transaction which wants to spend this script output must contain the actual datum. This option is the cheapest for the producing transaction because a hash is small and the transaction costs less fees. In this case the creator of the consuming transaction must know the value of the datum.

Producer Provides		Consumer Provides
In Output	In Tx Body	
Datum Hash	-	Datum
Datum Hash	Datum	Datum
(Inline) Datum	-	-

The second option is that beside the datum hash we include the actual datum in the transaction body, which means that people can look up the datum on the blockchain and then include it in their spending transaction. Such a producing transaction costs more than the previous one because it stores the datum in the transaction. However, it's not so convenient because the cardano node sees the statement value during validation but the forgets its value. So, for somebody else to later discover the datum on the blockchain another tool is needed like a chain indexer or dbsync. Since the Vasil hard fork there is also the third option where the output itself contains the datum. In this case we call it an inline datum and then it does not have to provide it in the body of the producing transaction. Additionally, the consuming transaction does not have to provide the datum and so it becomes smaller and cheaper.

Let's also now look at Cardano addresses in detail. A Cardano address has two parts. The payment part and optionally the staking part.



The payment part is responsible for deciding under which conditions an UTXO sitting at such an address can be spent. It is defined either by the hash of a public key or the hash of a plutus script. In Cardano we also call the private key the signing key and the public key the verification key. If it contains the public key hash UTXOs sitting at such an address can only be spent if the transaction is signed with the corresponding private key pair (signing key). But if it contains the script hash, the corresponding script is executed during validation and its output determines if one or more UTXOs sitting at the script address can be spent.

The optional staking part of an address decides who is entitled to staking rewards and is in control of delegation. Also, here you have two options. If the staking part is specified with a public key hash, then the owner of the corresponding private key is entitled to staking rewards. If it is a script hash, then the corresponding plutus script is executed for transactions that try to withdraw staking rewards for example.

Finally let's look at how the validation script can be referenced by the spending transaction. The spending or also called consuming transaction must provide the validation script as an input to the transaction. Because some scripts are used very often since the Vasil hard fork there is another way which is called reference scripts. In this case a plutus script can be attached to the datum of a UTXO. Usually, you do not want this UTXO to be consumed, so you can send it to a script address where the validation logic fails no matter the inputs. Then a spending transaction can simply reference this script sitting at a permanent UTXO instead of providing it as input. In either case a cardano node checks if the hash of the script equals the script address name which a spending transaction is processing. In the end we state that the redeemer is always supplied by the consuming transaction.

The EUTXO model is not tied to a specific programming language. What we have in Cardano is Plutus which is based on Haskell but you could use the same model with a different programming language. There are other blockchains also using EUTXO which are not using Plutus, as the Ergo blockchain for example.

1.5 Plutus code

The code for Plutus smart contracts can be separated into two parts. First is the "on-chain" code, which consists of the validator function and some additional declarations and variables as the script address. This code gets compiled to Plutus Core language or also called Plutus script. It runs on the Cardano blockchain and once submitted it cannot be changed.

From the official documentation [1] we get the following description for Plutus Core:

Plutus core is the scripting language used by Cardano to implement the EUTXO model. It is a simple, functional language similar to Haskell, and a large subset of Haskell can be used to write Plutus Core scripts. As a smart contract author, you don't write any Plutus Core; rather, all Plutus Core scripts are generated by a Haskell compiler plugin called Plutus Tx.

Community variants exist where you can write the on-chain code in another programming language and then it is compiled to Plutus core. We will talk about them at the end of this book.

The "off-chain" code can also be written in Haskell, just like the on-chain code. Unlike Ethereum where the on-chain code is written in Solidity, but the off-chain code is written in JavaScript.

That way, the business logic only needs to be written once. This logic can then be used in the validator script and in the code that builds the transactions that run the validator script.

The off-chain code basically constructs the transaction and submits it to the blockchain. Since both the on-chain and off-chain code are written in Haskell they can reside in one Haskell file while testing your code, which allows them to share code between them. Another option is also to construct the off-chain transaction with command line tools instead of compiling Haskell code. Cardano provides for this the *cardano-cli* command line tool. There are also community alternatives. Those tools will also be presented in later chapters of this book. In the 4th iteration of the Plutus pioneer program, we will not write any off-chain code but use only the command line tools. If you want to see examples of off-chain code written in Haskell, you can look at the videos of the 3rd PPP iteration or also read the accompanying book. However, some of the code examples may not work today since the Plutus libraries were upgraded in the meantime.

2 Resources

[1] Cardano Documentation

https://docs.cardano.org/plutus/learn-about-plutus

https://web.archive.org/web/20220704234049/https://docs.cardano.org/plutus/eutxo-explainer