Vexanium Blockchain Guide

BRIEF CONTENTS

GETTING STARTED

1. Foreword

1.1. Blockchain

Blockchain is an innovation in software protocol that at the first time founds its real-world massive application in Bitcoin that solved the double solving problem in digital currency.

A few years in the future, blockchain will be put into use because it has some characteristic that only unique in blockchain technology, such as:

- The data is copied in multiple nodes, it means if a hacker wants to attack, he needs to attack all of the nodes.
- Once stored in the blockchain, the data cannot be altered/edited
- Every block is connected to the previous block and the following block, it means if a hacker wants to attack or change the data, he needs to change the data in all of the block
- We can run a smart contract in the blockchain, that enables the creation of a new business model by elimination middleman or central authority

Based on the characters and core values that blockchain provides, here are some categorization (but not limited to) of how the blockchain technology solve problems in various industries :

1. Industries that need a faster value (example : money) transfer than traditional fiat money. Example : remittance, clearing and settlement of securities
2. Industries that involved many stakeholders that do not trust each other - or stakeholder are competing with each other but in some

situation also cooperate with each other. Example : Ports, Supply Chain.

3. Industries that are before blockchain need big money commitment to do transaction and because of it, the industries faced liquidity problem. For example to buy property or car need a lot of money. Now using blockchain technology, the property can be tokenized so buyer can buy the token with lower money commitment. This increase liquidity in the market. This is called “fractional ownership”.
4. Industries that needs immutability (can not be changed) character of the blockchain database, and the database can be accessed by all stakeholders. Example : public sector that deals with land registrations, education sector that deals with certificate.
5. Technology startup industries (example .social media, ecommerce marketplace, search engine, on demand startups) where the stakeholders think that they think they have a problem with the intermediary or middleman.
    a. The product they use have middleman with bargaining power that is too high, because this middleman got a lot of user data and sell it to advertisers (sometimes, illegally and without profit sharing with the users).
    b. Middlemen that take fees too high, from the users or merchants. And this fees can be reduced using blockchain & smart contracts - it reduced the cost of verification.
    c. Middlemen that is not really transparent about how things works : Games, Betting
6. Industries that have silo data sharing problem - sometimes before blockchain, the data is stored in various different institutions. Example : medical records, land registrations, Identity.

7. Industries that values digital content highly and need the digital content to be unique, secure and valuable in the digital era. Example : Games / digital goods, music/video, copyrighted content, digital collectibles, vote, carbon credit, social credit score.

1.2. Vexanium

Vexanium is the next generation blockchain software that is designed for Decentralized Apps (Dapps) Usability and retail penetration.

Vexanium focuses to solve the bottleneck and critical pain points of blockchain technology such as speed, scalability, usability and flexibility. The parallel processing technology and asynchronous communication methodology from Vexanium allow Dapps to operate and transact by processing simultaneously without having to increase load from the network.

Free transaction charges. Vexanium’s ownership structure allows free usage by the user and eliminates transaction charges. Blockchain project developers are allowed to use resources in proportion to their stake instead of the standard pay-per-transaction model

Vexanium blockchain architecture has the potential to scale to millions of transactions per second. Using Vexanium technology, the

deployment of the blockchain project will be faster, easier and more affordable.

(bacotan paragraf ini perlu ditambah, technology feature apa yang bikin potentially scale ?)

1.3. Technical Features

Vexanium blockchain uses C++ programming language for its smart contract programming language. C++ is the most studied programming languages in university and already has a lot of libraries - so developers don't have to "reinvent the wheel". C++ is known as "low-level language" that gives huge control for developers in terms of running the code and managing resources. C++ has superiority in speed, efficiency and safety and is widely used in "performance critical" application.

The C++ code is compiled in WebAssembly (WASM) and the smart contract is executed by WASM virtual machine.

WebAssembly is a low-level binary format for the web. It's not a programming language you are going to write.

https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts

https://flaviocopes.com/webassembly/

VEXIO uses C++ as its smart contract programming language. C++ is a popular programming language among developers around the world. Therefore, any developer who is familiar with C++ is not required to learn a new programming language and is more than ready to learn VEXIO's API, which will be covered in this onboarding series. Once

familiarity with VEXIO's API has been achieved, a developer will be able to program VEXIO smart contracts using C++.

Underlying VEXIO is a WebAssembly (WASM) virtual machine which executes smart contract code. WASM is also used by other important internet infrastructure software developed by Google, Microsoft, Apple, and others. The design choice of using WASM enables VEXIO to re-use optimized and battle-tested compilers and toolchains which are being maintained and improved by a broader community. In addition, adopting WASM standard also makes it easier for compiler developers to port other programming languages onto VEXIO.

1.4. Stack

2. Development Environment

2.1. Introduction

What you will learn in this chapter:

- How to quickly spin up nodes
- Manage wallets and keys
- Create accounts
- Write some contracts
- Compilation and ABI
- Deploy contracts

C/C++ experience

VEXIO based blockchains execute user-generated applications and code using WebAssembly (WASM). WASM is an emerging web standard

with widespread support from Google, Microsoft, Apple, and industry leading companies.

At the moment the most mature toolchain for building applications that compile to WASM is clang/llvm with their C/C++ compiler. For best compatibility, it is recommended that you use the VEXIO C++ toolchain.

Other toolchains in development by 3rd parties include: Rust, Python, and Solidity. While these other languages may appear simpler, their performance will likely impact the scale of application you can build. We expect that C++ will be the best language for developing high-performance and secure smart contracts and plan to use C++ for the foreseeable future.

Linux / Mac OS Experience

The VEXIO software supports the following environments:

- Amazon 2017.09 and higher
- Centos 7
- Fedora 25 and higher (Fedora 27 recommended)
- Mint 18
- Ubuntu 16.04 (Ubuntu 16.10 recommended)
- Ubuntu 18.04
- MacOS Darwin 10.12 and higher (MacOS 10.13.x recommended)

Command Line Knowledge

There are a variety of tools provided along with VEXIO which requires you to have basic command line knowledge in order to interact with.

C++ Environment Setup

We can use any text editor that, preferably, supports C++ syntax highlighting. Some of the popular editors are Sublime Text and Atom. Another option is an IDE, which provides a more sophisticated code completion and more complete development experience. You are welcome to use the software of your personal preference, but if you're unsure what to use we've provided some options for you to explore.

Potential Editors and IDEs

Operating System of Development Environment

If using an OS on any flavor of linux, you'll be able to follow these tutorials with ease, this includes but is not limited to

- Mac OS
- Ubuntu
- Debian

- Fedora

Windows

If you are developing on Windows, unfortunately we do not provided powershell ports and instructions at this time. In the future we may append powershell commands. In the mean-time your best bet is to use a VM with Ubuntu, and set up your development environment inside this VM. If you're an advanced Window's developer familiar with porting Linux instructions, you should encounter minimal issues.

2.2. Before You Start

Step 1: Install binaries

This tutorial is going to use pre-built binaries. For you to get started as quickly as possible this is the best option. Building from source is an option, but will set you back an hour or more and you may encounter build errors.

The below commands will download binaries for respective operating systems.

Mac OS X Brew Install:

```shell
brew tap vexio/vexio
brew install vexio
```

Ubuntu 18.04 Debian Package Install:

```shell
wget https://github.com/VEXIO/eos/releases/download/v1.7.0/vexio_1.7.0-1-ubuntu-18.04_amd64.deb
sudo apt install ./vexio_1.7.0-1-ubuntu-18.04_amd64.deb
```

Ubuntu 16.04 Debian Package Install:

```shell
wget https://github.com/VEXIO/eos/releases/download/v1.7.0/vexio_1.7.0-1-ubuntu-16.04_amd64.deb
sudo apt install ./vexio_1.7.0-1-ubuntu-16.04_amd64.deb
```

CentOS RPM Package Install:

```
Shell
```

```
wget https://github.com/VEXIO/eos/releases/download/v1.7.0/vexio-1.7.0-1.el7.x86_64.rpm
sudo yum install ./vexio-1.7.0-1.el7.x86_64.rpm
```

Fedora RPM Package Install

```
Text
```

```
wget https://github.com/VEXIO/eos/releases/download/v1.7.0/vexio-1.7.0-1.fc27.x86_64.rpm
sudo yum install ./vexio-1.7.0-1.fc27.x86_64.rpm
```

Step 2: Setup a development directory, stick to it.

You're going to need to pick a directory to work from, it's suggested to create a `contracts` directory somewhere on your local drive.

```
Shell
```

```
mkdir contracts
cd contracts
```

Step 3: Enter your local directory below.

Get the path of that directory and save it for later, as you're going to need it, you can use the following command to get your absolute path.

```
pwd
```

Enter the absolute path to your contract directory below, and it will be inserted throughout the documentation to make your life a bit easier. *This functionality requires cookies.*

2.3. Setup and Start Your Node

Step 1: Boot Node and Wallet

Step 1.1: Start keosd

First let us start keosd:

```
Shell
keosd &
```

You should see some output that looks like this:

```
Text
info  2018-11-26T06:54:24.789 thread-0  wallet_plugin.cpp:42
```

```
plugin_initialize    ] initializing wallet plugin
info  2018-11-26T06:54:24.795 thread-0  http_plugin.cpp:554
add_handler          ] add api url: /v1/keosd/stop
info  2018-11-26T06:54:24.796 thread-0  wallet_api_plugin.cpp:73
plugin_startup       ] starting wallet_api_plugin
info  2018-11-26T06:54:24.796 thread-0  http_plugin.cpp:554
add_handler          ] add api url: /v1/wallet/create
info  2018-11-26T06:54:24.796 thread-0  http_plugin.cpp:554
add_handler          ] add api url: /v1/wallet/create_key
info  2018-11-26T06:54:24.796 thread-0  http_plugin.cpp:554
add_handler          ] add api url: /v1/wallet/get_public_keys
```

Press enter to continue

Step 1.2: Start nodeos

Start nodeos now:

Shell

```
nodeos -e -p vexio \
--plugin vexio::producer_plugin \
--plugin vexio::chain_api_plugin \
--plugin vexio::http_plugin \
--access-control-allow-origin='*' \
--contracts-console \
--http-validate-host=false \
--verbose-http-errors >> nodeos.log 2>&1 &
```

These settings accomplish the following:

1. Run the Nodeos. This command loads all the basic plugins, set the server address, enable CORS and add some contract debugging and logging.
2. Enable CORS with no restrictions (*) and development logging

Step 2: Check the installation

Step 2.1: Check that Nodeos is Producing Blocks

Run the following command

Shell
tail -f nodeos.log

You should see some output in the console that looks like this:

Text
1929001ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366974ce4e2a... #13929 @ 2018-05-23T16:32:09.000 signed by vexio [trxs: 0, lib: 13928, confirmed: 0] 1929502ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366aea085023... #13930 @ 2018-05-23T16:32:09.500 signed by vexio [trxs: 0, lib: 13929, confirmed: 0] 1930002ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366b7f074fdd... #13931 @ 2018-05-23T16:32:10.000 signed by vexio [trxs: 0, lib: 13930, confirmed: 0] 1930501ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366cd8222adb... #13932 @ 2018-05-23T16:32:10.500 signed by vexio [trxs: 0, lib: 13931, confirmed: 0] 1931002ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366d5c1ec38d... #13933 @ 2018-05-23T16:32:11.000 signed by vexio [trxs: 0, lib: 13932, confirmed: 0] 1931501ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366e45c1f235... #13934 @ 2018-05-23T16:32:11.500 signed by vexio [trxs: 0, lib: 13933, confirmed: 0] 1932001ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 0000366f98adb324... #13935 @ 2018-05-23T16:32:12.000 signed by vexio [trxs: 0, lib: 13934, confirmed: 0] 1932501ms thread-0 producer_plugin.cpp:585 block_production_loo] Produced block 00003670a0f01daa... #13936 @

```
2018-05-23T16:32:12.500 signed by vexio [trxs: 0, lib: 13935, confirmed:
0]
1933001ms thread-0   producer_plugin.cpp:585
block_production_loo ] Produced block 00003671e8b36e1e... #13937 @
2018-05-23T16:32:13.000 signed by vexio [trxs: 0, lib: 13936, confirmed:
0]
1933501ms thread-0   producer_plugin.cpp:585
block_production_loo ] Produced block 0000367257fe1623... #13938 @
2018-05-23T16:32:13.500 signed by vexio [trxs: 0, lib: 13937, confirmed:
0]
```

Press `ctrl + c` to close the log

Step 2.2: Check the Wallet

Open the shell and run the cleos command to list available wallets. We will talk more about wallets in the future. For now, we need to validate the installation and see that the command line client

cleos is working as intended.

Shell
cleos wallet list

You should see a response with an empty list of wallets:

```
Wallets:

    []
```

From this point forward, you'll be executing commands from your local system (Linux or Mac)

Step 2.3: Check Nodeos endpoints

This will check that the RPC API is working correctly, pick one.

1. Check the `get_info` endpoint provided by the `chain_api_plugin` in your browser: http://localhost:8888/v1/chain/get_info
2. Check the same thing, but in the console on your **host machine**

Shell
curl http://localhost:8888/v1/chain/get_info

2.4. Install the CDT

The VEXIO Contract Development Toolkit, CDT for short, is a collection of tools related to contract compilation. Subsequent tutorials use the CDT primarily for compiling contracts and generating ABIs.

Starting from 1.3.x, CDT supports Mac OS X brew, Linux Debian and RPM packages. The easiest option to install would be using one of these package systems. Pick one installation method only.

Homebrew (Mac OS X)

Install

Shell
brew tap vexio/vexio.cdt brew install vexio.cdt

Uninstall

Shell
brew remove vexio.cdt

Ubuntu (Debian)

Install

Shell
wget https://github.com/VEXIO/vexio.cdt/releases/download/v1.6.1/vexio.cdt_1.6.1-1_amd64.deb sudo apt install ./vexio.cdt_1.6.1-1_amd64.deb

Uninstall

Shell
sudo apt remove vexio.cdt

CentOS/Redhat (RPM)

Install

Shell
wget https://github.com/VEXIO/vexio.cdt/releases/download/v1.6.1/vexio.cdt-1.6.1-1.centos-x86_64.rpm sudo yum install ./vexio.cdt-1.6.1-1.centos-x86_64.rpm

Uninstall

Shell
$ sudo yum remove vexio.cdt

Install from Source

The location where `vexio.cdt` is cloned is not that important because you will be installing `vexio.cdt`as a local binary in later steps. For now, you can clone `vexio.cdt` to your "contracts" directory previously created, or really anywhere else on your local system you see fit.

Text
cd CONTRACTS_DIR

Download

Clone version 1.6.1 of the `vexio.cdt` repository.

```
Text
git clone --recursive https://github.com/vexio/vexio.cdt --branch v1.6.1 --single-branch
cd vexio.cdt
```

It may take up to 30 minutes to clone the repository

Build

```
Shell
./build.sh
```

Install

```
Shell
sudo ./install.sh
```

The above command needs to be ran with `sudo` because `vexio.cdt`'s various binaries will be installed locally. You will be asked for your computer's account password.

Installing `vexio.cdt` will make the compiled binary global so it can be accessable anywhere. For this tutorial, **it is strongly suggested that you do not skip the install step for vexio.cdt**, failing to install will make it more difficult to follow this and other tutorials, and make usage in general more difficult.

Troubleshooting

Getting Errors during build.

- Search your errors for the string "/usr/local/include/vexiolib/"
- If found, `rm -fr /usr/local/include/vexiolib/` or navigate to `/usr/local/include/` and delete `vexiolib` using your operating system's file browser.

2.5. Create Development Wallet

Wallets are repositories of public-private key pairs. Private keys are needed to sign operations performed on the blockchain. Wallets are accessed using cleos.

Step 1: Create a Wallet

The first step is to create a wallet. Use cleos wallet create to create a new "default" wallet using the option `--to-console` for simplicity. If using

cleos in production, it's wise to instead use `--to-file` so your wallet password is not in your bash history. For development purposes and because these are **development and not production keys** `--to-console` poses no security threat.

Shell
cleos wallet create --to-console

`cleos` will return a password, save this password somewhere as you will likely need it later in the tutorial.

```
Creating wallet: default
Save password to use in the future to unlock this wallet.
Without password imported keys will not be retrievable.

    "PW5Kewn9L76X8Fpd.....................t42S9XCw2"
```

Step 2: Open the Wallet

Wallets are closed by default when starting a keosd instance, to begin, run the following

Shell
cleos wallet open

Run the following to return a list of wallets.

Text
cleos wallet list

and it will return

```
Wallets:
[
  "default"

    ]
```

Step 3: Unlock it

The `keosd` wallet(s) have been opened, but is still locked. Moments ago you were provided a password, you're going to need that now

Text
cleos wallet unlock

You will be prompted for your password, paste it and press enter.

Now run the following command

Text
cleos wallet list

It should now return

```
Wallets:
[
  "default *"

    ]
```

Pay special attention to the asterisk (*). This means that the wallet is currently **unlocked**

Step 4: Import keys into your wallet

Generate a private key, `cleos` has a helper function for this, just run the following.

Text
cleos wallet create_key

It will return something like..

```
    Created new private key with a public key of:
    "EOS8PEJ5FM42xLpHK...X6PymQu97KrGDJQY5Y"
```

Step 5: Follow this tutorial series more easily

Enter the public key provided in the last step in the box below. It will persist the **development public key**you just generated throughout the documentation.

Step 6: Import the Development Key

Every new VEXIO chain has a default "system" user called "vexio". This account is used to setup the chain by loading system contracts that dictate the governance and consensus of the VEXIO chain. Every new VEXIO chain comes with a development key, and this key is the same. Load this key to sign transactions on behalf of the system user (vexio)

Shell
cleos wallet import

You'll be prompted for a private key, enter the `vexio` development key provided below

5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3

Never use the development key for a production account! Doing so will most certainly result in the loss of access to your account, this private key is publicly known.

Wonderful, you now have a default wallet unlocked and loaded with a key, and are ready to proceed.

2.6. Create Test Accounts

What is an account?

An account is a collection of authorisations, stored on the blockchain, and used to identify a sender/recipient. It has a flexible authorisation structure that enables it to be owned either by an individual or group of individuals depending on **how**permissions have been configured. An account is required to send or receive a valid transaction to the blockchain

This tutorial series uses two "user" accounts, `bob` and `alice`, as well as the default `vexio` account for configuration. Additionally accounts are made for various contracts throughout this tutorial series.

Step 1: Create Test Accounts

In a previous step, you created a wallet and created a development key pair. You were asked to place that public key into a form, but either you skipped this step or have cookies disabled. You will need to replace YOUR_PUBLIC_KEY below with the public key you generated.

Throughout these tutorials the accounts `bob` and `alice` are used. Create two accounts using cleos create account

Shell
cleos create account vexio bob YOUR_PUBLIC_KEY cleos create account vexio alice YOUR_PUBLIC_KEY

You should then see a confirmation message similar to the following for each command that confirms that the transaction has been broadcast.

Result
executed transaction: 40c605006de... 200 bytes 153 us # vexio <= vexio::newaccount {"creator":"vexio","name":"alice","owner":{"threshold":1,"keys":[{"key":"EO S5rti4LTL53xptjgQBXv9HxyU... warning: transaction executed locally, but may not be confirmed by the network yet]

Step 2: Public Key

Note in `cleos` command a public key is associated with account `alice`. Each VEXIO account is associated with a public key.

Be aware that the account name is the only identifier for ownership. You can change the public key but it would not change the ownership of your VEXIO account.

Check which public key is associated with `alice` using cleos get account

Shell
cleos get account alice

You should see a message similar to the following:

Text
permissions: owner 1: 1 EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV active 1: 1 EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV memory: quota: unlimited used: 2.66 KiB net bandwidth: used: unlimited available: unlimited limit: unlimited cpu bandwidth: used: unlimited available: unlimited limit: unlimited

Notice that actually `alice` has both `owner` and `active` public keys. VEXIO has a unique authorization structure that has added security for you account. You can minimize the exposure of your account by keeping the owner key cold, while using the key associated with your `active`

permission. This way, if your `active` key were ever compromised, you could regain control over your account with your `owner` key.

In term of authorization, if you have a `owner` permission you can change the private key of `active` permission. But you cannot do so other way around.

Troubleshooting

If you get an error while creating the account, make sure your wallet is unlocked

Shell
cleos wallet list

You should see an asterisk (*) next to the wallet name, as seen below.

Text
Wallets: ["default *"]

3. Smart Contract Development

3.1. Start Your World

Smart contracts are lines of code that describe business logic. In essence, it facilitates 3 functions:

- Storing business rules
- Verifying rules
- Being self-executable.

In explaining smart contracts, usually using the analogy of a vending machine, when the user enters 1 USD to the vending machine, the user can choose snacks worth 1 USD from the vending machine, after the user selects the snack of his choice, the vending machine will give out the snack of the users choice.

A smart contract can execute itself where the agreement/agreement between the buyer and seller has been written in the code sequence. These codes and agreements are stored on a blockchain network that is distributed and decentralized.

With the format of money that is natively digital or digital currency, the smart contract allows transactions and agreements to be carried out between parties who do not know each other, do not trust each other without a central authority, legal system, external enforcement mechanism, or middlemen.

A smart contract makes transactions become traceable, transparent, and irreversible (cannot be cancelled). Smart contract eliminates middlemen fees, so transactions will be cheaper and faster because it removes the cost of verification and audit.

Smart contract characteristics:

- No intermediaries
- No lawyer
- No escrow agent

Create a new directory called "hello" in the contracts directory you previously created, or through your system GUI or with CLI and enter the directory.

```
Shell
cd CONTRACTS_DIR
mkdir hello
cd hello
```

Create a new file "hello.cpp" using the command “touch”, and open it in your preffered editor.

```
Shell
touch hello.cpp
```

Below the `vexio.hpp` header file is included. The `vexio.hpp` file includes a few classes required to write a smart contract.

```
C++
#include <vexio/vexio.hpp>
```

Namespace in C++ is a declarative region in which you can put (or separate) many functions in it. Using the `vexio` namespace will reduce clutter in your code. For example, by setting `using namespace vexio;`, `vexio::print("foo")` can be written `print("foo")`

```
C++
using namespace vexio;
```

Create a standard C++11 class. The contract class needs to extend `vexio::contract` class which is included earlier from the vexio.hpp header. In the code below we dont use [[vexio::contract]] because we already put vexio.hpp in include header.

```
C++
#include <vexio/vexio.hpp>

using namespace vexio;

class hello : public contract {};
```

An empty contract doesn't do much good. Add a public access specifier and a using-declaration. The `using` declaration will allow us to write more concise code.

```
C++
#include <vexio/vexio.hpp>

using namespace vexio;

class hello : public contract {
  public:
          using contract::contract;
};
```

In this contract example, we will print “hello world” and then the name of user (which we assume only “user”). So we write an action that accepts a "name" parameter, and then prints that parameter out. Actions implement the behaviour of a contract.

C++

```
#include <vexio/vexio.hpp>

using namespace vexio;

class hello : public contract {
  public:
     using contract::contract;

     void hi( name user ) {
        print( "Hello, ", user);
     }
};
```

Typedef is a feature in C++ to define explicitly new data type names . Using **typedef** does not actually create a new data class, rather it defines a new name for an existing type.

The above action accepts a parameter called `user` that's a `name` type. VEXIO comes with a number of typedefs, one of the most common typedefs you'll encounter is `name`. Using the `vexio::print` library previously included in vexio.hpp, concatenate a string and print the `user` parameter. Use the braced initialization of `name{user}` to make the `user` parameter printable.

Everything together, here's the completed hello world contract

C++

```
#include <vexio/vexio.hpp>

using namespace vexio;

class hello : public contract {
  public:
     using contract::contract;

     void hi( name user ) {
        print( "Hello, ", user);
     }
};
```

You can compile your code to web assembly (.wasm) as follows:

Shell

```
vexio-cpp hello.cpp -o hello.wasm
```

When a contract is deployed, it is deployed to an account, and the account becomes the interface for the contract. As mentioned earlier these tutorials use the same public key for all of the accounts to keep things simple.

Shell

```
cleos wallet keys
```

Create an account for the contract using cleos create account, with the command provided below.

Shell
cleos create account vexio hello YOUR_PUBLIC_KEY -p vexio@active

Deploy the compiled `wasm` to the blockchain with cleos set contract.

In previous steps you should have created a `contracts` directory and obtained the absolute path and then saved it into a cookie. Replace "CONTRACTS_DIR" in the command below with the absolute path to your `contracts` directory.

Shell
cleos set contract hello CONTRACTS_DIR/hello -p hello@active

Great! Now the contract is set, push an action to it.

Shell
cleos push action hello hi '["bob"]' -p bob@active

Result
executed transaction: 4c10c1426c16b1656e802f3302677594731b380b18a44851d38e8b5275072857 244 bytes 1000 cycles # hello.code <= hello.code::hi {"user":"bob"} >> Hello, bob

As written, the contract will allow any account to say hi to any user

Shell

```
cleos push action hello hi '["bob"]' -p alice@active
```

Result

```
executed transaction:
28d92256c8ffd8b0255be324e4596b7c745f50f85722d0c4400471bc184b
9a16  244 bytes  1000 cycles
#    hello.code <= hello.code::hi              {"user":"bob"}
>> Hello, bob
```

As expected, the console output is "Hello, bob"

In this case "alice" is the one who authorized it and `user` is just an argument. Modify the contract so that the authorizing user, "alice" in this case, must be the same as the user the contract is responding "hi" to. Use the `require_auth` method. This method takes a `name` as a parameter, and will check if the user executing the action matches the provided parameter.

C++

```
void hi( name user ) {
   require_auth( user );
   print( "Hello, ", name{user} );
}
```

Recompile the contract

Shell
vexio-cpp -abigen -o hello.wasm hello.cpp

And then update it

Shell
cleos set contract hello CONTRACTS_DIR/hello -p hello@active

Try to execute the action again, but this time with mismatched authorization.

Shell
cleos push action hello hi '["bob"]' -p alice@active

As expected, `require_auth` halted the transaction and threw an error.

Result
Error 3090004: Missing required authority Ensure that you have the related authority inside your transaction!; If you are currently using 'cleos push action' command, try to add the relevant authority using -p option.

Now, with our change, the contract verifies the provided `name user` is the same as the authorising user. Try it again, but this time, with the authority of the "alice" account.

```
Text
cleos push action hello hi '["alice"]' -p alice@active
```

```
Text
executed transaction:
235bd766c2097f4a698cfb948eb2e709532df8d18458b92c9c6aae74ed8e
4518  244 bytes  1000 cycles
#    hello <= hello::hi              {"user":"alice"}
>> Hello, alice
```

3.2. Deploy, Issue, and Transfer Token

Step 1: Obtain Contract Source

Navigate to your contracts directory.

```
Text
cd CONTRACTS_DIR
```

Pull the source

```
Text
git clone https://github.com/VEXIO/vexio.contracts --branch v1.5.2
--single-branch
```

This repository contains several contracts, but it's the `vexio.token` contract that is important now. Navigate to the directory now.

```
Text
cd vexio.contracts/vexio.token
```

Step 2: Create Account for Contract

Before we can deploy the token contract we must create an account to deploy it to, we'll use the **vexio development key** for this account.

```
Shell
cleos create account vexio vexio.token
EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV
```

Step 3: Compile the Contract

```
Shell
vexio-cpp -I include -o vexio.token.wasm src/vexio.token.cpp --abigen
```

Step 4: Deploy the Token Contract

```
Shell
```

```
cleos set contract vexio.token
CONTRACTS_DIR/vexio.contracts/vexio.token --abi vexio.token.abi
-p vexio.token@active
```

Result

```
Reading WASM from ...
Publishing contract...
executed transaction:
69c68b1bd5d61a0cc146b11e89e11f02527f24e4b240731c4003ad1dc
0c87c2c  9696 bytes  6290 us
#         vexio <= vexio::setcode
{"account":"vexio.token","vmtype":0,"vmversion":0,"code":"0061736d0
100000001aa011c60037f7e7f0060047f...
#         vexio <= vexio::setabi
{"account":"vexio.token","abi":"0e656f73696f3a3a6162692f312e3000
0605636c6f73650002056f776e6572046e61...
warning: transaction executed locally, but may not be confirmed by
the network yet         ]
```

Step 5: Create the Token

To create a new token call the `create(...)` action with the proper arguments. This action accepts 1 argument, it's a `symbol_name` type composed of two pieces of data, a maximum supply float and a `symbol_name` in capitalized alpha characters only, for example "1.0000 SYS". The issuer will be the one with authority to call issue and or perform other actions such as freezing, recalling, and whitelisting of owners.

Below is the concise way to call this method, using positional arguments:

Shell

```
cleos push action vexio.token create '[ "vexio", "1000000000.0000
SYS"]' -p vexio.token@active
```

Result

```
executed transaction:
0e49a421f6e75f4c5e09dd738a02d3f51bd18a0cf31894f68d335cd70d
9c0e12  120 bytes  1000 cycles
#   vexio.token <= vexio.token::create
{"issuer":"vexio","maximum_supply":"1000000000.0000 SYS"}
```

An alternate approach uses named arguments:

Shell

```
cleos push action vexio.token create '{"issuer":"vexio",
"maximum_supply":"1000000000.0000 SYS"}' -p vexio.token@active
```

Result

```
executed transaction:
0e49a421f6e75f4c5e09dd738a02d3f51bd18a0cf31894f68d335cd70d
9c0e12  120 bytes  1000 cycles
#   vexio.token <= vexio.token::create
{"issuer":"vexio","maximum_supply":"1000000000.0000 SYS"}
```

This command created a new token `SYS` with a precision of 4 decimals and a maximum supply of 1000000000.0000 SYS. To create this token requires the permission of the `vexio.token` contract. For this reason, `-p vexio.token@active` was passed to authorize the request.

Step 6: Issue Tokens

The issuer can issue new tokens to the "alice" account created earlier.

Text

```
cleos push action vexio.token issue '[ "alice", "100.0000 SYS", "memo" ]'
```

```
-p vexio@active
```

Result

```
executed transaction:
822a607a9196112831ecc2dc14ffb1722634f1749f3ac18b73ffacd41160b
019  268 bytes  1000 cycles
#   vexio.token <= vexio.token::issue
{"to":"user","quantity":"100.0000 SYS","memo":"memo"}
>> issue
#   vexio.token <= vexio.token::transfer
{"from":"vexio","to":"user","quantity":"100.0000 SYS","memo":"memo"}
>> transfer
#         vexio <= vexio.token::transfer
{"from":"vexio","to":"user","quantity":"100.0000 SYS","memo":"memo"}
#          user <= vexio.token::transfer
{"from":"vexio","to":"user","quantity":"100.0000 SYS","memo":"memo"}
```

This time the output contains several different actions: one issue and three transfers. While the only action signed was `issue`, the `issue` action performed an "inline transfer" and the "inline transfer" notified the sender and receiver accounts. The output indicates all of the action handlers that were called, the order they were called in, and whether or not any output was generated by the action.

Technically, the `vexio.token` contract could have skipped the `inline transfer` and opted to just modify the balances directly. However, in this case the `vexio.token` contract is following our token convention that requires that all account balances be derivable by the sum of the transfer actions that reference them. It also requires that the sender and receiver of funds be notified so they can automate handling deposits and withdrawals.

To inspect the transaction, try using the `-d -j` options, they indicate "don't broadcast" and "return transaction as json," which you may find useful during development.

Shell

```
cleos push action vexio.token issue '["alice", "100.0000 SYS", "memo"]'
-p vexio@active -d -j
```

Step 7: Transfer Tokens

Now that account `alice` has been issued tokens, transfer some of them to account `bob`. It was previously indicated that `alice` authorized this action using the argument `-p alice@active`.

Shell

```
cleos push action vexio.token transfer '[ "alice", "bob", "25.0000 SYS",
"m" ]' -p alice@active
```

Result

```
executed transaction:
06d0a99652c11637230d08a207520bf38066b8817ef7cafaab2f0344aafd
7018  268 bytes  1000 cycles
#   vexio.token <= vexio.token::transfer
{"from":"alice","to":"bob","quantity":"25.0000 SYS","memo":"Here you go
bob!"}
>> transfer
#         user <= vexio.token::transfer
{"from":"alice","to":"bob","quantity":"25.0000 SYS","memo":"Here you go
bob!"}
#       tester <= vexio.token::transfer
{"from":"alice","to":"bob","quantity":"25.0000 SYS","memo":"Here you go
bob!"}
```

Now check if "bob" got the tokens using cleos get currency balance

Shell

```
cleos get currency balance vexio.token bob SYS
```

Text
25.00 SYS

Check "alice's" balance, notice that tokens were deducted from the account

Shell
cleos get currency balance vexio.token alice SYS

Text
75.00 SYS

3.3. ABI Files

Introduction

Previously you deployed the `vexio.token` contract using the provided ABI file. This tutorial will overview how the ABI file correlates to the `vexio.token` contract.

ABI files can be generated using the `vexio-cpp` utility provided by `vexio.cdt`. However, there are several situations that may cause ABI's generation to malfunction or fail altogether. Advanced C++ patterns can trip it up and custom types can sometimes cause issues for ABI generation. For this reason, it's **imperative** you understand how ABI files work, so you can debug and fix if and when necessary.

What is an ABI?

The Application Binary Interface (ABI) is a JSON-based description on how to convert user actions between their JSON and Binary representations. The ABI also describes how to convert the database state to/from JSON. Once you have described your contract via an ABI then developers and users will be able to interact with your contract seamlessly via JSON.

Create an ABI File

Start with an empty ABI, name it `vexio.token.abi`

Text

```
{
  "version": "vexio::abi/1.0",
  "types": [],
  "structs": [],
  "actions": [],
  "tables": [],
  "ricardian_clauses": [],
  "abi_extensions": [],
  "___comment" : ""
}
```

Types

An ABI enables any client or interface to interpret and even generate a GUI for your contract. For this to work in a consistent manner,

describe the custom types that are used as a parameter in any public action or struct that needs to be described in the ABI.

JSON

```
{
  "new_type_name": "name",
  "type": "name"
}
```

The ABI now looks like this:

JSON

```
{
  "version": "vexio::abi/1.0",
  "types": [{
    "new_type_name": "name",
    "type": "name"
          }],
  "structs": [],
  "actions": [],
  "tables": [],
  "ricardian_clauses": [],
  "abi_extensions": []
}
```

Structs

Structs that are exposed to the ABI also need to be described. By looking at vexio.token.hpp, it can be quickly determined which structs are utilized by public actions. This is particularly important for the next step.

A struct's object definition in JSON looks like the following:

JSON

```
{
   "name": "issue", //The name
   "base": "",                          //Inheritance, parent struct
   "fields": []                         //Array of field objects describing the
struct's fields.
}
```

Fields

JSON

```
{
   "name":"", // The field's name
   "type":""   // The field's type
}
```

In the `vexio.token` contract, there's a number of structs that require definition. Please note, not all of the structs are explicitly defined, some correspond to an actions' parameters. Here's a list of structs that require an ABI description for the `vexio.token` contract:

Implicit Structs

The following structs are implicit in that a struct was never explicitly defined in the contract. Looking at the create action, you'll find two parameters, `issuer` of type `name` and `maximum_supply` of type `asset`. For

brevity this tutorial won't break down every struct, but applying the same logic, you will end up with the following:

create

JSON

```
{
  "name": "create",
  "base": "",
  "fields": [
    {
      "name":"issuer",
      "type":"name"
    },
    {
      "name":"maximum_supply",
      "type":"asset"
    }
  ]
}
```

issue

JSON

```
{
  "name": "issue",
  "base": "",
  "fields": [
    {
      "name":"to",
      "type":"name"
    },
    {
      "name":"quantity",
```

```
      "type":"asset"
    },
    {
      "name":"memo",
      "type":"string"
    }
  ]
}
```

retire

JSON

```
{
  "name": "retire",
  "base": "",
  "fields": [
    {
      "name":"quantity",
      "type":"asset"
    },
    {
      "name":"memo",
      "type":"string"
    }
  ]
}
```

transfer

JSON

```
{
```

```
  "name": "transfer",
  "base": "",
  "fields": [
    {
      "name":"from",
      "type":"name"
    },
    {
      "name":"to",
      "type":"name"
    },
    {
      "name":"quantity",
      "type":"asset"
    },
    {
      "name":"memo",
      "type":"string"
    }
  ]
}
```


close

JSON


```
{
  "name": "close",
  "base": "",
  "fields": [
    {
      "name":"owner",
      "type":"name"
    },
    {
      "name":"symbol",
      "type":"symbol"
    }
  ]
 }
```

Explicit Structs

These structs are explicitly defined, as they are a requirement to instantiate a multi-index table. Describing them is no different than defining the implicit structs as demonstrated above.

account

JSON

```
{
  "name": "account",
  "base": "",
  "fields": [
    {
      "name":"balance",
      "type":"asset"
    }
  ]
}
```

currency_stats

JSON

```
{
  "name": "currency_stats",
  "base": "",
  "fields": [
    {
      "name":"supply",
      "type":"asset"
    },
```

```
   {
     "name":"max_supply",
     "type":"asset"
   },
   {
     "name":"issuer",
     "type":"account_name"
   }
 ]
}
```

Actions

An action's JSON object definition looks like the following:

JSON

```
{
  "name": "transfer",                    //The name of the action as defined in the contract
  "type": "transfer",                    //The name of the implicit struct as described in the ABI
  "ricardian_contract": ""          //An optional ricardian clause to associate to this action describing its intended functionality.
}
```

Describe the actions of the `vexio.token` contract by aggregating all the public functions described in the `vexio.token`contract's header file.

Then describe each action's *type* according to its previously described struct. In most situations, the function name and the struct name will be equal, but are not required to be equal.

Below is a list of actions that link to their source code with example JSON provided for how each action would be described.

create

JSON

```
{
  "name": "create",
  "type": "create",
  "ricardian_contract": ""
}
```

issue

JSON

```
{
  "name": "issue",
  "type": "issue",
  "ricardian_contract": ""
}
```

retire

JSON

```
{
  "name": "retire",
  "type": "retire",
  "ricardian_contract": ""
}
```

transfer

JSON

```
{
  "name": "transfer",
  "type": "transfer",
  "ricardian_contract": ""
}
```

close

JSON

```
{
  "name": "close",
  "type": "close",
  "ricardian_contract": ""
}
```

Tables

Describe the tables. Here's a table's JSON object definition:

JSON

```
{
  "name": "",       //The name of the table, determined during instantiation.
  "type": "",                              //The table's corresponding struct
  "index_type": "", //The type of primary index of this table
  "key_names" : [], //An array of key names, length must equal length of
```

```
key_types member
  "key_types" : []  //An array of key types that correspond to key names array member, length of array must equal length of key names array.
}
```

The vexio.token contract instantiates two tables, accounts and stats.

The accounts table is an i64 index, based on the `account` struct, has a `uint64` as it's primary key

Here's how the accounts table would be described in the ABI

JSON

```
{
  "name": "accounts",
  "type": "account", // Corresponds to previously defined struct
  "index_type": "i64",
  "key_names" : ["primary_key"],
  "key_types" : ["uint64"]
}
```

The stat table is an i64 index, based on the `currency_stats` struct, has a `uint64` as it's primary key

Here's how the stat table would be described in the ABI

JSON

```
{
  "name": "stat",
  "type": "currency_stats",
  "index_type": "i64",
  "key_names" : ["primary_key"],
  "key_types" : ["uint64"]
}
```

You'll notice the above tables have the same "key name." Naming your keys similar names is symbolic in that it can potentially suggest a subjective relationship. As with this implementation, implying that any given value can be used to query different tables.

Putting it all Together

Finally, an ABI file that accurately describes the `vexio.token` contract.

JSON

```
{
  "version": "vexio::abi/1.0",
  "types": [
    {
      "new_type_name": "name",
      "type": "name"
    }
  ],
  "structs": [
    {
      "name": "create",
      "base": "",
      "fields": [
        {
          "name":"issuer",
          "type":"name"
```

```
    },
    {
      "name":"maximum_supply",
      "type":"asset"
    }
  ]
},
{
  "name": "issue",
  "base": "",
  "fields": [
    {
      "name":"to",
      "type":"name"
    },
    {
      "name":"quantity",
      "type":"asset"
    },
    {
      "name":"memo",
      "type":"string"
    }
  ]
},
{
  "name": "retire",
  "base": "",
  "fields": [
    {
      "name":"quantity",
      "type":"asset"
    },
    {
      "name":"memo",
      "type":"string"
    }
  ]
},
{
  "name": "close",
  "base": "",
  "fields": [
    {
      "name":"owner",
      "type":"name"
    },
    {
      "name":"symbol",
```

```
        "type":"symbol"
      }
    ]
  },
  {
    "name": "transfer",
    "base": "",
    "fields": [
      {
        "name":"from",
        "type":"name"
      },
      {
        "name":"to",
        "type":"name"
      },
      {
        "name":"quantity",
        "type":"asset"
      },
      {
        "name":"memo",
        "type":"string"
      }
    ]
  },
  {
    "name": "account",
    "base": "",
    "fields": [
      {
        "name":"balance",
        "type":"asset"
      }
    ]
  },
  {
    "name": "currency_stats",
    "base": "",
    "fields": [
      {
        "name":"supply",
        "type":"asset"
      },
      {
        "name":"max_supply",
        "type":"asset"
      },
      {
```

```
        "name":"issuer",
        "type":"name"
      }
    ]
  }
],
"actions": [
  {
    "name": "transfer",
    "type": "transfer",
    "ricardian_contract": ""
  },
  {
    "name": "issue",
    "type": "issue",
    "ricardian_contract": ""
  },
  {
    "name": "retire",
    "type": "retire",
    "ricardian_contract": ""
  },
  {
    "name": "create",
    "type": "create",
    "ricardian_contract": ""
  },
  {
    "name": "close",
    "type": "close",
    "ricardian_contract": ""
  }
],
"tables": [
  {
    "name": "accounts",
    "type": "account",
    "index_type": "i64",
    "key_names" : ["currency"],
    "key_types" : ["uint64"]
  },
  {
    "name": "stat",
    "type": "currency_stats",
    "index_type": "i64",
    "key_names" : ["currency"],
    "key_types" : ["uint64"]
  }
],
```

```
  "ricardian_clauses": [],
  "abi_extensions": []
}
```

Cases not Covered by Token Contract

Vectors

When describing a vector in your ABI file, simply append the type with `[]`, so if you need to describe a vector of permission levels, you would describe it like so: `permission_level[]`

Struct Base

It's a rarely used property worth mentioning. You can use **base** ABI struct property to reference another struct for inheritance, as long as that struct is also described in the same ABI file. Base will do nothing or potentially throw an error if your smart contract logic does not support inheritance.

You can see an example of base in use in the system contract source code and ABI

Extra ABI Properties Not Covered Here

A few properties of the ABI specification were skipped here for brevity, however, there is a pending ABI specification that will outline every property of the ABI in its entirety.

Ricardian Clauses

Ricardian clauses describe the intended outcome of a particular actions. It may also be utilized to establish terms between the sender and the contract.

ABI Extensions

A generic "future proofing" layer that allows old clients to skip the parsing of "chunks" of extension data. For now, this property is unused. In the future each extension would have its own "chunk" in that vector so that older clients skip it and newer clients that understand how to interpret it.

Maintenance

Every time you change a struct, add a table, add an action or add parameters to an action, use a new type, you will need to remember to update your ABI file. In many cases failure to update your ABI file will not produce any error.

Troubleshooting

Table returns no rows

Check that your table is accurately described in the ABI file. For example, If you use `cleos` to add a table on a contract with a malformed ABI definition and then get rows from that table, you will receive an empty result. `cleos` will not produce an error when adding a row nor reading a row when a contract has failed to properly describe its tables in its ABI.

3.4. Data Persistence

To learn about data persistence, write a simple smart contract that functions as an address book. While this use case isn't very practical as a production smart contract for various reasons, it's a good contract to start with to learn how data persistence works on VEXIO without being distracted by business logic that does not pertain to vexio's `multi_index` functionality.

Step 1: Create a new directory

Earlier, you created a contract directory, navigate there now.

Shell
cd CONTRACTS_DIR

Create a new directory for our contract and enter the directory

C++

```
mkdir addressbook
cd addressbook
```

Step 2: Create and open a new file

C++

```
touch addressbook.cpp
```

open the file in your favorite editor.

Step 3: Write an Extended Standard Class and Include VEXIO

In a previous tutorial, you created a hello world contract and you learned the basics. You will be familiar with the structure below, the class has been named `addressbook` respectively.

C++

```
#include <vexio/vexio.hpp>

using namespace vexio;

class [[vexio::contract("addressbook")]] addressbook : public
vexio::contract {
  public:

  private:

};
```

Step 4: Create The Data Structure for the Table

Before a table can be configured and instantiated, a struct that represents the data structure of the address book needs to be written. Since it's an address book, the table will contain people, so create a `struct` called "person"

C++
struct person {};

When defining the structure of a multi_index table, you will require a unique value to use as the primary key.

For this contract, use a field called "key" with type `name`. This contract will have one unique entry per user, so this key will be a consistent and guaranteed unique value based on the user's `name`

C++

struct person {
 name key;
};

Since this contract is an address book it probably should store some relevant details for each entry or *person*

C++

```
struct person {
 name key;
 std::string first_name;
 std::string last_name;
 std::string street;
 std::string city;
 std::string state;
};
```

Great. The basic data structure is now complete.

Next, define a `primary_key` method. Every multi_index struct requires a *primary key* to be set. Behind the scenes, this method is used according to the index specification of your multi_index instantiation. VEXIO wraps boost::multi_index

Create an method `primary_key()` and return a struct member, in this case, the `key` member as previously discussed.

C++

```
struct person {
 name key;
 std::string first_name;
 std::string last_name;
 std::string street;
 std::string city;
 std::string state;

 uint64_t primary_key() const { return key.value;}
};
```

Step 5: Configure the Multi-Index Table

Now that the data structure of the table has been defined with a `struct` we need to configure the table. The vexio::multi_index constructor needs to be named and configured to use the struct we previously defined.

C++
typedef vexio::multi_index<"people"_n, person> address_index;

With the above `multi_index` configuration there is a table named **people**, that

1. Uses the _n operator to define an vexio::name type and uses that to name the table. This table contains a number of different singular "persons", so name the table "people".
2. Pass in the singular `person` struct defined in the previous step.
3. Declare this table's type. This type will be used to instantiate this table later.
4. There are some additional configurations, such as configuring indices, that will be covered further on.

So far, our file should look like this.

C++
#include <vexio/vexio.hpp> using namespace vexio; class [[vexio::contract("addressbook")]] addressbook : public vexio::contract {

```
  public:

  private:
    struct [[vexio::table]] person {
      name key;
      std::string first_name;
      std::string last_name;
      std::string street;
      std::string city;
      std::string state;

      uint64_t primary_key() const { return key.value;}
    };

    typedef vexio::multi_index<"people"_n, person> address_index;
};
```

Step 6: The Constructor

When working with C++ classes, the first public method you should create is a constructor.

Our constructor will be responsible for initially setting up the contract.

VEXIO contracts extend the *contract* class. Initialize our parent *contract* class with the code name of the contract and the receiver. The important parameter here is the `code` parameter which is the account on the blockchain that the contract is being deployed to.

C++
addressbook(name receiver, name code, datastream<const char*>

```
ds):contract(receiver, code, ds) {}
```

Step 7: Adding a record to the table

Previously, the primary key of the multi-index table was defined to enforce that this contract will only store one record per user. To make it all work, some assumptions about the design need to be established.

1. The only account authorized to modify the address book is the user.
2. the **primary_key** of our table is unique, based on username
3. For usability, the contract should have the ability to both create and modify a table row with a single action.

In vexio a chain has unique accounts, so `name` is an ideal candidate as a **primary_key** in this specific use case. The nametype is a `uint64_t`.

Next, define an action for the user to add or update a record. This action will need to accept any values that this action needs to be able to emplace (create) or modify.

For user-experience and interface simplicity, have a single method be responsible for both creation and modification of rows. Because of this behavior, name it "upsert," a combination of "update" and "insert."

```
C++
void upsert(
```

```
  name user,
  std::string first_name,
  std::string last_name,
  std::string street,
  std::string city,
  std::string state
) {}
```

Earlier, it was mentioned that only the user has control over their own record, as this contract is opt-in. To do this, utilize the require_auth method provided by the `vexio.cdt`. This method accepts an `name` type argument and asserts that the account executing the transaction equals the provided value and has the proper permissions to do so.

C++

```
void upsert(name user, std::string first_name, std::string last_name,
std::string street, std::string city, std::string state) {
  require_auth( user );
}
```

Previously, a multi_index table was configured, and declared as `address_index`. To instantiate a table, two parameters are required:

1. The first parameter "code", which specifies the owner of this table. As the owner, the account will be charged for storage costs. Also, only that account can modify or delete the data in

this table unless another payer is specified. Here we use the `get_self()` function which will pass the name of this contract.

2. The second parameter "scope" which ensures the uniqueness of the table within this contract. In this case, since we only have one table we can use the value from `get_first_receiver()`. *`get_first_receiver` is the account name this contract is deployed to.*

Note that scopes are used to logically separate tables within a multi-index (see the vexio.token contract multi-index for an example, which scopes the table on the token owner). Scopes were originally intended to separate table state in order to allow for parallel computation on the individual sub-tables. However, currently inter-blockchain communication has been prioritized over parallelism. Because of this, scopes are currently only used to logically separate the tables as in the case of vexio.token.

C++

```
void upsert(name user, std::string first_name, std::string last_name,
std::string street, std::string city, std::string state) {
  require_auth( user );
  address_index addresses(get_self(), get_first_receiver().value);
}
```

Next, query the iterator, setting it to a variable since this iterator will be used several times.

C++

```
void upsert(name user, std::string first_name, std::string last_name,
std::string street, std::string city, std::string state) {
  require_auth( user );
  address_index addresses(get_self(), get_first_receiver().value);
  auto iterator = addresses.find(user.value);
}
```

Security has been established and the table instantiated, great! Next up, write the code for creating or modifying the table.

First, detect whether or not a particular user already exists in the table. To do this, use table's find method by passing the `user` parameter. The find method will return an iterator. Use that iterator to test it against the end method. The "end" method is an alias for "null".

C++

```
void upsert(name user, std::string first_name, std::string last_name,
std::string street, std::string city, std::string state) {
  require_auth( user );
  address_index addresses(get_self(), get_first_receiver().value);
  auto iterator = addresses.find(user.value);
  if( iterator == addresses.end() )
  {
    //The user isn't in the table
  }
  else {
    //The user is in the table
  }
}
```

Create a record in the table using the multi_index method emplace. This method accepts two arguments, the "payer" of this record who pays the storage usage and a callback function.

The callback function for the emplace method must use a lamba function to create a reference. Inside the body assign the row's values with the ones provided to `upsert`.

C++

```
void upsert(name user, std::string first_name, std::string last_name,
std::string street, std::string city, std::string state) {
  require_auth( user );
  address_index addresses(get_self(), get_first_receiver().value);
  auto iterator = addresses.find(user.value);
  if( iterator == addresses.end() )
  {
    addresses.emplace(user, [&]( auto& row ) {
      row.key = user;
      row.first_name = first_name;
      row.last_name = last_name;
      row.street = street;
      row.city = city;
      row.state = state;
    });
  }
  else {
    //The user is in the table
  }
}
```

Next, handle the modification, or update, case of the "upsert" function. Use the modify method, passing a few arguments:

- The iterator defined earlier, presently set to the user as declared when calling this action.

- The "payer", who will pay for the storage cost of this row, in this case, the user.
- The callback function that actually modifies the row

C++

```
void upsert(name user, std::string first_name, std::string last_name,
std::string street, std::string city, std::string state) {
  require_auth( user );
  address_index addresses(get_self(), get_first_receiver().value);
  auto iterator = addresses.find(user.value);
  if( iterator == addresses.end() )
  {
    addresses.emplace(user, [&]( auto& row ) {
      row.key = user;
      row.first_name = first_name;
      row.last_name = last_name;
      row.street = street;
      row.city = city;
      row.state = state;
    });
  }
  else {
    addresses.modify(iterator, user, [&]( auto& row ) {
      row.key = user;
      row.first_name = first_name;
      row.last_name = last_name;
      row.street = street;
      row.city = city;
      row.state = state;
    });
  }
}
```

The `addressbook` contract now has a functional action that will enable a user to create a row in the table if that record does not yet exist, and modify it if it already exists.

But what if the user wants to remove the record entirely?

Step 8: Remove record from the table

Similar to the previous steps, create a public method in the `addressbook`, making sure to include the ABI declarations and a require_auth that tests against the action's argument `user` to verify only the owner of a record can modify their account.

C++

```
   void erase(name user){
     require_auth(user);
   }
```

Instantiate the table. In `addressbook` each account has only one record. Set `iterator` with find

C++

```
...
   void erase(name user){
     require_auth(user);
     address_index addresses(get_self(), get_first_receiver().value);
     auto iterator = addresses.find(user.value);
   }
...
```

Finally, call the erase method, to erase the iterator. Once the row is erased, the storage space will be free up for the original payer.

C++

```
...
  void erase(name user) {
    require_auth(user);
    address_index addresses(get_self(), get_first_receiver().value);
    auto iterator = addresses.find(user.value);
    check(iterator != addresses.end(), "Record does not exist");
    addresses.erase(iterator);
  }
...
```

The contract is now mostly complete. Users can create, modify and erase records. However, the contract is not quite ready to be compiled.

Step 9: Preparing for the ABI

9.1 ABI Action Declarations

vexio.cdt includes an ABI Generator, but for it to work will require some declarations.

Above both the `upsert` and `erase` functions add the following C++11 declaration:

C++

```
[[vexio::action]]
```

The above declaration will extract the arguments of the action and create necessary ABI *struct* descriptions in the generated ABI file.

9.2 ABI Table Declarations

Add an ABI declaration to the table. Modify the following line defined in the private region of your contract:

```
C++
struct person {
```

To this:

```
C++
struct [[vexio::table]] person {
```

The `[[vexio.table]]` declaration will add the necessary descriptions to the ABI file.

Now our contract is ready to be compiled.

Below is the final state of our `addressbook` contract:

```
C++
#include <vexio/vexio.hpp>
```

```
using namespace vexio;

class [[vexio::contract("addressbook")]] addressbook : public
vexio::contract {

public:

  addressbook(name receiver, name code,  datastream<const
char*> ds): contract(receiver, code, ds) {}

  [[vexio::action]]
  void upsert(name user, std::string first_name, std::string
last_name, std::string street, std::string city, std::string
state) {
    require_auth( user );
    address_index addresses( get_self(),
get_first_receiver().value );
    auto iterator = addresses.find(user.value);
    if( iterator == addresses.end() )
    {
      addresses.emplace(user, [&]( auto& row ) {
       row.key = user;
       row.first_name = first_name;
       row.last_name = last_name;
       row.street = street;
       row.city = city;
       row.state = state;
      });
    }
    else {
      addresses.modify(iterator, user, [&]( auto& row ) {
        row.key = user;
        row.first_name = first_name;
        row.last_name = last_name;
        row.street = street;
        row.city = city;
        row.state = state;
      });
    }
  }

  [[vexio::action]]
  void erase(name user) {
    require_auth(user);

    address_index addresses( get_self(),
get_first_receiver().value);

    auto iterator = addresses.find(user.value);
    check(iterator != addresses.end(), "Record does not
exist");
    addresses.erase(iterator);
  }

private:
  struct [[vexio::table]] person {
    name key;
    std::string first_name;
    std::string last_name;
```

```
    std::string street;
    std::string city;
    std::string state;
    uint64_t primary_key() const { return key.value; }
  };
  typedef vexio::multi_index<"people"_n, person>
address_index;

};
```

Step 10 Prepare the Ricardian Contract [Optional]

To define a Ricardian contract for this smart contract create and open a new file called addressbook.contracts.md. Notice that the name of the Ricardian contract must match the name of the smart contract, in this case `addressbook`.

Shell

```
touch addressbook.contracts.md
```

Add Ricardian Contract definitions to this file.

YAML

```
<h1 class="contract">upsert</h1>
---
spec-version: 0.0.2
title: Upsert
summary: This action will either insert or update an entry in the address
book. If an entry exists with the same name as the specified user
parameter, the record is updated with the first_name, last_name, street,
```

```
city, and state parameters. If a record does not exist, a new record is created. The data is stored in the multi index table. The ram costs are paid by the smart contract.
icon:

<h1 class="contract">erase</h1>
---
spec-version: 0.0.2
title: Erase
summary: This action will remove an entry from the address book if an entry in the multi index table exists with the specified name.
icon:
```

Contracts compiled without a Ricardian contract will generate a compiler warning for each action missing an entry in the Ricardian contract.

```
Warning, action <action_name> does not have a ricardian contract
```

Step 11 Prepare the Ricardian Clauses [Optional]

To define Ricardian clauses for this smart contract create and open a new file called addressbook.clauses.md. Notice again that the name of the Ricardian clauses must match the name of the smart contract.

Shell
touch addressbook.clauses.md

Add Ricardian clause definitions to this file:

YAML

```
<h1 class="clause">Data Storage</h1>
---
spec-version: 0.0.1
title: General Data Storage
summary: This smart contract will store data added by the user. The user
consents to the storage of this data by signing the transaction.
icon:


<h1 class="clause">Data Usage</h1>
---
spec-version: 0.0.1
title: General Data Use
summary: This smart contract will store user data. The smart contract will
not use the stored data for any purpose outside store and delete.
icon:

<h1 class="clause">Data Ownership</h1>
---
spec-version: 0.0.1
title: Data Ownership
summary: The user of this smart contract verifies that the data is owned
by the smart contract, and that the smart contract can use the data in
accordance to the terms defined in the Ricardian Contract.
icon:

<h1 class="clause">Data Distirbution</h1>
---
spec-version: 0.0.1
title: Data Ownership
summary: The smart contract promises to not actively share or distribute
the address data. The user of the smart contract understands that data
stored in a multi index table is not private data and can be accessed by
any user of the blockchain.
icon:


<h1 class="clause">Data Future</h1>
---
spec-version: 0.0.1
title: Data Ownership
summary: The smart contract promises to only use the data in
accordance of the terms defined in the Ricardian Contract, now and at all
```

future dates. icon:

Contracts compiled without a Ricardian clause will generate a compiler warning for each action missing an entry in the Ricardian clause.

```
Warning, empty ricardian clause file
```

Step 12: Compile the Contract

Execute the following command from your terminal.

Shell
vexio-cpp addressbook.cpp -o addressbook.wasm

If you created a Ricardian contract and Ricardian clauses, the definitions will appear appear in the .abi file. An example for the addressbook.cpp, built including the contract and . clause definitions described above is shown below.

Text

```
{
   "____comment": "This file was generated with vexio-abigen. DO NOT EDIT ",
   "version": "vexio::abi/1.1",
   "types": [],
   "structs": [
      {
         "name": "erase",
```

```
      "base": "",
      "fields": [
        {
          "name": "user",
          "type": "name"
        }
      ]
    },
    {
      "name": "person",
      "base": "",
      "fields": [
        {
          "name": "key",
          "type": "name"
        },
        {
          "name": "first_name",
          "type": "string"
        },
        {
          "name": "last_name",
          "type": "string"
        },
        {
          "name": "street",
          "type": "string"
        },
        {
          "name": "city",
          "type": "string"
        },
        {
          "name": "state",
          "type": "string"
        }
      ]
    },
    {
      "name": "upsert",
      "base": "",
      "fields": [
        {
          "name": "user",
          "type": "name"
        },
        {
          "name": "first_name",
          "type": "string"
```

```
            },
            {
               "name": "last_name",
               "type": "string"
            },
            {
               "name": "street",
               "type": "string"
            },
            {
               "name": "city",
               "type": "string"
            },
            {
               "name": "state",
               "type": "string"
            }
         ]
      }
   ],
   "actions": [
      {
         "name": "erase",
         "type": "erase",
         "ricardian_contract": "---\nspec-version: 0.0.2\ntitle:
Erase\nsummary: his action will remove an entry from the address book
if an entry exists with the same name \nicon:"
      },
      {
         "name": "upsert",
         "type": "upsert",
         "ricardian_contract": "---\nspec-version: 0.0.2\ntitle:
Upsert\nsummary: This action will either insert or update an entry in the
address book. If an entry exists with the same name as the user
parameter the record is updated with the first_name, last_name, street,
city and state parameters. If a record does not exist a new record is
created. The data is stored in the multi index table. The ram costs are
paid by the smart contract.\nicon:"
      }
   ],
   "tables": [
      {
         "name": "people",
         "type": "person",
         "index_type": "i64",
         "key_names": [],
         "key_types": []
      }
   ],
```

```
    "ricardian_clauses": [
        {
            "id": "Data Storage",
            "body": "---\nspec-version: 0.0.1\ntitle: General data
Storage\nsummary: This smart contract will store data added by the
user. The user verifies they are happy for this data to be stored.\nicon:"
        },
        {
            "id": "Data Usage",
            "body": "---\nspec-version: 0.0.1\ntitle: General data
Use\nsummary: This smart contract will store user data. The smart
contract will not use the stored data for any purpose outside store and
delete \nicon:"
        },
        {
            "id": "Data Ownership",
            "body": "---\nspec-version: 0.0.1\ntitle: Data
Ownership\nsummary: The user of this smart contract verifies that the
data is owned by the smart contract, and that the smart contract can use
the data in accordance to the terms defined in the Ricardian Contract
\nicon:"
        },
        {
            "id": "Data Distirbution",
            "body": "---\nspec-version: 0.0.1\ntitle: Data
Ownership\nsummary: The smart contract promises to not actively share
or distribute the address data. The user of the smart contract
understands that data stored in a multi index table is not private data and
can be accessed by any user of the blockchain.  \nicon:"
        },
        {
            "id": "Data Future",
            "body": "---\nspec-version: 0.0.1\ntitle: Data
Ownership\nsummary: The smart contract promises to only use the data
in accordance to the terms defined in the Ricardian Contract, now and at
all future dates. \nicon:"
        }
    ],
    "variants": []
}
```


Step 13: Deploy the Contract

Create an account for the contract, execute the following shell command

Shell
cleos create account vexio addressbook YOUR_PUBLIC_KEY YOUR_PUBLIC_KEY -p vexio@active

Deploy the `addressbook` contract

Text
struct person {};

Result
5f78f9aea400783342b41a989b1b4821ffca006cd76ead38ebdf97428559daa0 5152 bytes 727 us # vexio <= vexio::setcode {"account":"addressbook","vmtype":0,"vmversion":0,"code":"0061736d010000000191011760077f7e7f7f7f7f7f... # vexio <= vexio::setabi {"account":"addressbook","abi":"0e656f73696f3a3a6162692f312e30010c6163636f756e745f6e616d65046e616d65... warning: transaction executed locally, but may not be confirmed by the network yet]

Step 14: Test the Contract

Add a row to the table

Shell
cleos push action addressbook upsert '["alice", "alice", "liddell", "123 drink me way", "wonderland", "amsterdam"]' -p alice@active

Result
executed transaction: 003f787824c7823b2cc8210f34daed592c2cfa66cbbfd4b904308b0dfeb0c811 152 bytes 692 us # addressbook <= addressbook::upsert {"user":"alice","first_name":"alice","last_name":"liddell","street":"123 drink me way","city":"wonde...

Check that **alice** cannot add records for another user.

Text
cleos push action addressbook upsert '["bob", "bob", "is a loser", "doesnt exist", "somewhere", "someplace"]' -p alice@active

As expected, the `require_auth` in our contract prevented alice from creating/modifying another user's row.

Result
Error 3090004: Missing required authority Ensure that you have the related authority inside your transaction!; If you are currently using 'cleos push action' command, try to add the relevant authority using -p option.

Retrieve alice's record.

Shell
cleos get table addressbook addressbook people --lower alice --limit 1

Result
{ "rows": [{ "key": "3773036822876127232", "first_name": "alice", "last_name": "liddell", "street": "123 drink me way", "city": "wonderland", "state": "amsterdam" }], "more": false }

Test to see that **alice** can remove the record.

Shell
cleos push action addressbook erase '["alice"]' -p alice@active

Result
executed transaction: 0a690e21f259bb4e37242cdb57d768a49a95e39a83749a02bced652ac4b3f4ed 104 bytes 1623 us

```
#   addressbook <= addressbook::erase           {"user":"alice"}
warning: transaction executed locally, but may not be confirmed by the
network yet    ]
```

Check that the record was removed:

Shell

```
cleos get table addressbook addressbook people --lower alice --limit 1
```

Result

```
{
  "rows": [],
  "more": false
}
```

Looking good!

Wrapping Up

You've learned how to configure tables, instantiate tables, create new rows, modify existing rows and work with iterators. You've learned how to test against an empty iterator result. Congrats!

3.5. Secondary Indices

VEXIO has the ability to sort tables by up to 16 indices. In the following section, we're going to add another index to the `addressbook` contract, so we can iterate through the records in a different way.

Step 1: Remove existing data from table

As mentioned earlier, *a table's struct cannot be modified when it contains data.* The first step allows the removal of the data already added.

Remove all records of alice and bob that were added in previous tutorials.

Shell
cleos push action addressbook erase '["alice"]' -p alice@active

Shell
cleos push action addressbook erase '["bob"]' -p bob@active

Step 2: Add new index member and getter

Add a new member variable and its getter to the `addressbook.cpp` contract. Since the secondary index needs to be numeric field, a `uint64_t` age variable is added.

C++

```
uint64_t age;
uint64_t get_secondary_1() const { return age;}
```

Step 3: Add secondary index to `addresses` table configuration

A field has been defined as the secondary index, next the `address_index` table needs to be reconfigured.

C++

```
typedef vexio::multi_index<"people"_n, person,
indexed_by<"byage"_n, const_mem_fun<person, uint64_t,
&person::get_secondary_1>>
  > address_index;
```

In the third parameter, we pass a `index_by` struct which is used to instantiate a index.

In that `index_by` struct, we specify the name of index as `"byage"` and the second type parameter as a function call operator which extracts a const value as an index key. In this case, we point it to the getter we created earlier so this multiple index table will index records by the `age` variable.

C++

```
indexed_by<"byage"_n, const_mem_fun<person, uint64_t, &person::get_secondary_1>>
```

Step 4: Modify code

With all the changes in previous steps, we can now update the `upsert` function. Change the function parameter list to the following:

C++

```
  void upsert(name user, std::string first_name, std::string last_name, uint64_t age, std::string street, std::string city, std::string state)
```

Add additional lines to update `age` field in `upsert` function as the following:

C++

```
  void upsert(name user, std::string first_name, std::string last_name, uint64_t age, std::string street, std::string city, std::string state) {
    require_auth( user );
    address_index addresses( get_first_receiver(), get_first_receiver().value);
    auto iterator = addresses.find(user.value);
    if( iterator == addresses.end() )
    {
      addresses.emplace(user, [&]( auto& row ) {
       row.key = user;
       row.first_name = first_name;
       row.last_name = last_name;
       // -- Add code below --
       row.age = age;
```

```
      row.street = street;
      row.city = city;
      row.state = state;
    });
  }
  else {
    addresses.modify(iterator, user, [&]( auto& row ) {
      row.key = user;
      row.first_name = first_name;
      row.last_name = last_name;
      // -- Add code below --
      row.age = age;
      row.street = street;
      row.city = city;
      row.state = state;
    });
  }
}
```

Step 5: Compile and Deploy

Compile

```
Shell
vexio-cpp --abigen addressbook.cpp -o addressbook.wasm
```

Deploy

```
Shell
cleos set contract addressbook CONTRACTS_DIR/addressbook
```

Step 6: Test it

Insert records

```
Shell
cleos push action addressbook upsert '["alice", "alice", "liddell", 9, "123 drink me way", "wonderland", "amsterdam"]' -p alice@active
```


```
Shell
cleos push action addressbook upsert '["alice", "alice", "liddell", 9, "123 drink me way", "wonderland", "amsterdam"]' -p alice@active
```


Look up alice's address by the age index. Here the `--index 2` parameter is used to indicate that the query applies to the secondary index (index #2)

```
Shell
cleos get table addressbook addressbook people --upper 10 \
--key-type i64 \
--index 2
```

You should see something like the following

```
JSON
{
  "rows": [{
      "key": "alice",
      "first_name": "alice",
      "last_name": "liddell",
```

```
      "age": 9,
      "street": "123 drink me way",
      "city": "wonderland",
      "state": "amsterdam"
    }
  ],
  "more": false
}
```

Look it up by Bob's age

Shell

```
cleos get table addressbook addressbook people --upper 50 --key-type i64 --index 2
```

It should return

JSON

```
{
  "rows": [{
      "key": "alice",
      "first_name": "alice",
      "last_name": "liddell",
      "age": 9,
      "street": "123 drink me way",
      "city": "wonderland",
      "state": "amsterdam"
    },{
      "key": "bob",
      "first_name": "bob",
      "last_name": "is a loser",
      "age": 49,
      "street": "doesnt exist",
      "city": "somewhere",
      "state": "someplace"
    }
  ],
```

```
  "more": false
}
```

Wrapping Up

The complete `addressbook` contract up to this point:

C++

```
#include <vexio/vexio.hpp>
#include <vexio/print.hpp>

using namespace vexio;

class [[vexio::contract("addressbook")]] addressbook : public
vexio::contract {

public:

  addressbook(name receiver, name code,  datastream<const char*> ds):
contract(receiver, code, ds) {}

  [[vexio::action]]
  void upsert(name user, std::string first_name, std::string last_name,
uint64_t age, std::string street, std::string city, std::string state) {
    require_auth( user );
    address_index
addresses(get_first_receiver(),get_first_receiver().value);
    auto iterator = addresses.find(user.value);
    if( iterator == addresses.end() )
    {
      addresses.emplace(user, [&]( auto& row ) {
       row.key = user;
       row.first_name = first_name;
       row.last_name = last_name;
       row.age = age;
       row.street = street;
       row.city = city;
       row.state = state;
      });
    }
    else {
```

```
      addresses.modify(iterator, user, [&]( auto& row ) {
        row.key = user;
        row.first_name = first_name;
        row.last_name = last_name;
        row.age = age;
        row.street = street;
        row.city = city;
        row.state = state;
      });
    }
  }

  [[vexio::action]]
  void erase(name user) {
    require_auth(user);

    address_index addresses(get_self(), get_first_receiver().value);

    auto iterator = addresses.find(user.value);
    check(iterator != addresses.end(), "Record does not exist");
    addresses.erase(iterator);
  }

private:
  struct [[vexio::table]] person {
    name key;
    std::string first_name;
    std::string last_name;
    uint64_t age;
    std::string street;
    std::string city;
    std::string state;

    uint64_t primary_key() const { return key.value; }
    uint64_t get_secondary_1() const { return age; }

  };

  typedef vexio::multi_index<"people"_n, person, indexed_by<"byage"_n,
const_mem_fun<person, uint64_t, &person::get_secondary_1>>>
address_index;

};
```

3.6. Adding Inline Action

Introduction

It was previously demonstrated by authoring the `addressbook` contract the basics of multi-index tables. In this part of the series you'll learn how to construct actions, and send those actions from within a contract.

Step 1: Adding vexio.code to permissions

In order for the inline actions to be sent from `addressbook`, add the `vexio.code` permission to the contract's account's active permission. Open your terminal and run the following code

Shell
cleos set account permission addressbook active --add-code

The `vexio.code` authority is a pseudo authority implemented to enhance security, and enable contracts to execute inline actions.

Step 2: Notify Action

If not still opened, open the `addressbook.cpp` contract authored in the last tutorial. Write an action that dispatches a "transaction receipt"

whenever a transaction occurs. To do this, create a helper function in the `addressbook` class.

C++

```
[[vexio::action]]
void notify(name user, std::string msg) {}
```

This function is very simple, it just accepts a user account as a `name` type and a message as a `string` type. The user parameter dictates which user gets the message that is sent.

Step 3: Copy action to sender using require_recipient

This transaction needs to be copied to the user so it can be considered as a receipt. To do this, use the require_recipientmethod. Calling `require_recipient` adds an account to the require_recipient set and ensures that these accounts receive a notification of the action being executed. The notification is like sending a "carbon copy" of the action to the accounts in the require_recipient set.

C++

```
[[vexio::action]]
void notify(name user, std::string msg) {
 require_recipient(user);
}
```

This action is very simple, however, as written, any user could call this function, and "fake" a receipt from this contract. This could be used in

malicious ways, and should be seen as a vulnerability. To correct this, require that the authorization provided in the call to this action is from the contract itself, for this, use get_self

C++

```
[[vexio::action]]
void notify(name user, std::string msg) {
  require_auth(get_self());
  require_recipient(user);
}
```

Now if user `bob` calls this function directly, but passes the parameter `alice` the action will throw an exception.

Step 4: Notify helper for sending inline transactions

Since this inline action will be called several times, write a quick helper for maximum code reuse. In the private region of your contract, define a new method.

C++

```
...
 private:
   void send_summary(name user, std::string message){}
```

Inside of this helper construct an action and send it.

Step 5: The Action Constructor

Modify the `addressbook` contract to send a receipt to the user every time they take an action on the contract.

To begin, address the "create record" case. This is the case that fires when a record is not found in the table, i.e., when `iterator == addresses.end()` is `true`.

Save this object to an `action` variable called `notification`

C++

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        //permission_level,
        //code,
        //action,
        //data
      );
    }
```

The action constructor requires a number of parameters.

- A permission_level struct
- The contract to call (initialised using `vexio::name` type)
- The action (initialised using `vexio::name` type)
- The data to pass to the action, a tuple of positionals that correlate to the actions being called.

The Permission struct

In this contract the permission should be authorized by the `active` authority of the contract using `get_self()`. As a reminder, to use the 'active`authority inline you will need your contract's to give active authority to`vexio.code` pseudo-authority (instructions above)

C++

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        permission_level{get_self(),"active"_n},
      );
    }
```

The "code" AKA "account where contract is deployed"

Since the action called is in this contract, use get_self. `"addressbook"_n` would also work here, but if this contract were deployed under a different account name, it wouldn't work. Because of this, `get_self()` is the superior option.

C++

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        permission_level{get_self(),"active"_n},
        get_self(),
        //action
        //data
      );
    }
```

The action

The `notify` action was previously defined to be called from this inline action. Use the _n operator here.

C++

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        permission_level{get_self(),"active"_n},
        get_self(),
        "notify"_n,
        //data
      );
    }
```

The Data

Finally, define the data to pass to this action. The notify function accepts two parameters, an `name` and a `string`. The action constructor expects data as type `bytes`, so use `make_tuple`, a function available through `std` C++ library. Data passed in the tuple is positional, and determined by the order of the parameters accepted by the action that being called.

- Pass the `user` variable that is provided as a parameter of the `upsert()` action.
- Concatenate a string that includes the name of the user, and include the `message` to pass to the `notify` action.

C++

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        permission_level{get_self(),"active"_n},
        get_self(),
        "notify"_n,
        std::make_tuple(user, name{user}.to_string() + message)
      );
    }
```

Send the action.

Finally, send the action using the `send` method of the action struct.

C++

```
...
  private:
    void send_summary(name user, std::string message){
      action(
        permission_level{get_self(),"active"_n},
        get_self(),
        "notify"_n,
        std::make_tuple(user, name{user}.to_string() + message)
      ).send();
    }
```

Step 6: Call the helper and inject relevant messages.

Now that the helper is defined, it should probably be called from the relevant locations. There's three specific places for the new `notify` helper to be called from:

- After the contract `emplaces` a new record: `send_summary(user, "successfully emplaced record to addressbook");`
- After the contract `modifies` an existing record: `send_summary(user, "successfully modified record in addressbook.");`
- After the contract `erases` an existing record: `send_summary(user, "successfully erased record from addressbook");`

Step 7: Recompile and Regenerate the ABI File

Now that everything is in place, here's the current state of the `addressbook` contract:

C++

```
#include <vexio/vexio.hpp>
#include <vexio/print.hpp>

using namespace vexio;

class [[vexio::contract("addressbook")]] addressbook : public vexio::contract {

public:

  addressbook(name receiver, name code,  datastream<const char*> ds): contract(receiver, code, ds) {}

  [[vexio::action]]
  void upsert(name user, std::string first_name, std::string last_name, uint64_t age, std::string street, std::string city, std::string state) {
```

```
    require_auth(user);
    address_index addresses(get_first_receiver(),
get_first_receiver().value);
    auto iterator = addresses.find(user.value);
    if( iterator == addresses.end() )
    {
      addresses.emplace(user, [&]( auto& row ) {
       row.key = user;
       row.first_name = first_name;
       row.last_name = last_name;
       row.age = age;
       row.street = street;
       row.city = city;
       row.state = state;
      });
      send_summary(user, " successfully emplaced record to
addressbook");
    }
    else {
      addresses.modify(iterator, user, [&]( auto& row ) {
        row.key = user;
        row.first_name = first_name;
        row.last_name = last_name;
        row.street = street;
        row.city = city;
        row.state = state;
      });
      send_summary(user, " successfully modified record to
addressbook");
    }
  }

  [[vexio::action]]
  void erase(name user) {
    require_auth(user);

    address_index addresses(get_first_receiver(),
get_first_receiver().value);

    auto iterator = addresses.find(user.value);
    check(iterator != addresses.end(), "Record does not exist");
    addresses.erase(iterator);
    send_summary(user, " successfully erased record from
addressbook");
  }
```

```
  [[vexio::action]]
  void notify(name user, std::string msg) {
    require_auth(get_self());
    require_recipient(user);
  }

private:
  struct [[vexio::table]] person {
    name key;
    std::string first_name;
    std::string last_name;
    uint64_t age;
    std::string street;
    std::string city;
    std::string state;

    uint64_t primary_key() const { return key.value; }
    uint64_t get_secondary_1() const { return age;}
  };

  void send_summary(name user, std::string message) {
    action(
      permission_level{get_self(),"active"_n},
      get_self(),
      "notify"_n,
      std::make_tuple(user, name{user}.to_string() + message)
    ).send();
  };


  typedef vexio::multi_index<"people"_n, person,
    indexed_by<"byage"_n, const_mem_fun<person, uint64_t,
&person::get_secondary_1>>
  > address_index;

};
```

Open your terminal, and navigate to `CONTRACTS_DIR/addressbook`

Shell

```
cd CONTRACTS_DIR/addressbook
```

Now, recompile the contract, including the `--abigen` flag since changes have been made to the contract that affects the ABI. If you've followed the instructions carefully, you shouldn't see any errors.

Shell
vexio-cpp -o addressbook.wasm addressbook.cpp --abigen

Smart contracts on VEXIO are upgradeable so the contract can be redeployed with changes.

Shell
cleos set contract addressbook CONTRACTS_DIR/addressbook

Result
Publishing contract... executed transaction: 1898d22d994c97824228b24a1741ca3bd5c7bc2eba9fea8e83446d78bfb264fd 7320 bytes 747 us #   vexio <= vexio::setcode {"account":"addressbook","vmtype":0,"vmversion":0,"code":"0061736d0100000001a6011a60027f7e0060077f7e... #   vexio <= vexio::setabi {"account":"addressbook","abi":"0e656f73696f3a3a6162692f312e30010c6163636f756e745f6e616d65046e616d65...

Success!

Step 8: Testing it

Now that the contract has been modified and deployed, test it. In the previous tutorial, alice's addressbook record was deleted during the

testing steps, so calling `upsert` will fire the inline action just written inside of the "create" case.

Run the following command in your terminal

Shell
cleos push action addressbook upsert '["alice", "alice", "liddell", 21, "123 drink me way", "wonderland", "amsterdam"]' -p alice@active

`cleos` will return some data, that includes all the actions executed in the transaction.

Shell
executed transaction: e9e30524186bb6501cf490ceb744fe50654eb393ce0dd733f3bb6c68ff4b5622 160 bytes 9810 us # addressbook <= addressbook::upsert {"user":"alice","first_name":"alice","last_name":"liddell","age":21,"street":"123 drink me way","cit... # addressbook <= addressbook::notify {"user":"alice","msg":"alicesuccessfully emplaced record to addressbook"} # alice <= addressbook::notify {"user":"alice","msg":"alicesuccessfully emplaced record to addressbook"}

The last entry in the previous log is an `addressbook::notify` action sent to `alice`. Use cleos get actions to display actions executed and relevant to alice.

Shell
cleos get actions alice

Result

```
# seq  when                          contract::action => receiver      trx id...
args
=============================================================================
======
#   62   2018-09-15T12:57:09.000      addressbook::notify => alice
685ecc09... {"user":"alice","msg":"alice successfully added record to
ad...
```

3.7. Inline Action to External Contract

Previously, we sent an inline action to an action that was defined in the contract. In this part of the tutorial, we'll explore sending actions to an external contract. Since we've already gone over quite a bit of contract authoring, we'll keep this contract extremely simple. We'll author a contract that counts actions written by the contract. This contract has very little real-world use, but will demonstrate inline action calls to an external contract

Step 1: The Addressbook Counter Contract

Navigate to `CONTRACTS_DIR` if not already there, create a directory called `abcounter` and then create a `abcounter.cpp` file

Shell

```
cd CONTRACTS_DIR
mkdir abcounter
```

```
touch abcounter.cpp
```

Open the `abcounter.cpp` file in your favorite editor and paste the following code into the file. This contract is very basic, and for the most part does not cover much that we haven't already covered up until this point. There are a few exceptions though, and they are covered in full below.

C++

```
#include <vexio/vexio.hpp>

using namespace vexio;

class [[vexio::contract("abcounter")]] abcounter : public vexio::contract {
  public:

    abcounter(name receiver, name code,  datastream<const char*> ds):
contract(receiver, code, ds) {}

    [[vexio::action]]
    void count(name user, std::string type) {
      require_auth( name("addressbook"));
      count_index counts(get_first_receiver(), get_first_receiver().value);
      auto iterator = counts.find(user.value);

      if (iterator == counts.end()) {
        counts.emplace("addressbook"_n, [&]( auto& row ) {
          row.key = user;
          row.emplaced = (type == "emplace") ? 1 : 0;
          row.modified = (type == "modify") ? 1 : 0;
          row.erased = (type == "erase") ? 1 : 0;
        });
      }
      else {
        counts.modify(iterator, "addressbook"_n, [&]( auto& row ) {
          if(type == "emplace") { row.emplaced += 1; }
          if(type == "modify") { row.modified += 1; }
          if(type == "erase") { row.erased += 1; }
        });
      }
    }

    using count_action = action_wrapper<"count"_n, &abcounter::count>;
```

```
  private:
    struct [[vexio::table]] counter {
      name key;
      uint64_t emplaced;
      uint64_t modified;
      uint64_t erased;
      uint64_t primary_key() const { return key.value; }
    };

    using count_index = vexio::multi_index<"counts"_n, counter>;
};
```

The first new concept in the code above is that we are explicitly restricting calls to the one action to a **specific account** in this contract using require_auth to the `addressbook` contract, as seen below.

C++

```
//Only the addressbook account/contract can authorize this command.
require_auth( name("addressbook"));
```

Previously, a dynamic value was used with `require_auth`.

Another new concept in the code above, is action wrapper. As shown below the first template parameter is the 'action' we are going to call and the second one should point to the action function

Text

```
using count_action = action_wrapper<"count"_n, &abcounter::count>;
```

Step 2: Create Account for abcounter Contract

Open your terminal and execute the following command to create the **abcounter** user.

Shell
cleos create account vexio abcounter YOUR_PUBLIC_KEY

Step 3: Compile and Deploy

Shell
vexio-cpp -o abcounter.wasm abcounter.cpp

Finally, deploy the `abcounter` contract.

Shell
cleos set contract abcounter CONTRACTS_DIR/abcounter

Step 4: Modify addressbook contract to send inline-action to abcounter

Navigate to your addressbook directory now.

Shell
cd CONTRACTS_DIR/addressbook

Open the `addressbook.cpp` file in your favorite editor if not already open.

In the last part of this series, we went over inline actions to our own contract. This time, we are going to send an inline action to another contract, our new `abcounter` contract.

Create another helper called `increment_counter` under the `private` declaration of the contract as below:

C++

```
void increment_counter(name user, std::string type) {
    abcounter::count_action count("abcounter"_n, {get_self(), "active"_n});
    count.send(user, type);
}
```

Let's go through the code listing above.

This time we use the action wrapper instead of calling a function. To do that, we firstly initialised the count_action object defined earlier. The first parameter we pass is the callee contract name, in this case `abcounter`. The second parameter is the permission struct.

- For the permission, get_self() returns the current `addressbook` contract. The `active` permission of `addressbook` is used.

Unlike the `Adding Inline Actions` tutorial, we won't need to specify the action because the action wrapper type incorporates the action when it is defined.

In line 3 we call the action with the data, namely `user` and `type` which are required by the `abcounter` contract.

Now, add the following calls to the helpers in their respective action scopes.

Text

```
//Emplace
increment_counter(user, "emplace");
//Modify
increment_counter(user, "modify");
//Erase
increment_counter(user, "erase");
```

Now your `addressbook.cpp` contract should look like this.

C++

```
#include <vexio/vexio.hpp>
#include "abcounter.cpp"

using namespace vexio;

class [[vexio::contract("addressbook")]] addressbook : public
vexio::contract {

public:

  addressbook(name receiver, name code,  datastream<const char*> ds):
contract(receiver, code, ds) {}
```

```
  [[vexio::action]]
  void upsert(name user, std::string first_name, std::string last_name,
      uint64_t age, std::string street, std::string city, std::string state) {
    require_auth(user);
    address_index addresses(get_first_receiver(),
get_first_receiver().value);
    auto iterator = addresses.find(user.value);
    if( iterator == addresses.end() )
    {
      addresses.emplace(user, [&]( auto& row ) {
       row.key = user;
       row.first_name = first_name;
       row.last_name = last_name;
       row.age = age;
       row.street = street;
       row.city = city;
       row.state = state;
      });
      send_summary(user, " successfully emplaced record to
addressbook");
      increment_counter(user, "emplace");
    }
    else {
      std::string changes;
      addresses.modify(iterator, user, [&]( auto& row ) {
        row.key = user;
        row.first_name = first_name;
        row.last_name = last_name;
        row.age = age;
        row.street = street;
        row.city = city;
        row.state = state;
      });
      send_summary(user, " successfully modified record to
addressbook");
      increment_counter(user, "modify");
    }
  }

  [[vexio::action]]
  void erase(name user) {
    require_auth(user);

    address_index addresses(get_first_receiver(),
get_first_receiver().value);

    auto iterator = addresses.find(user.value);
    check(iterator != addresses.end(), "Record does not exist");
    addresses.erase(iterator);
```

```
    send_summary(user, " successfully erased record from
addressbook");
    increment_counter(user, "erase");
  }

  [[vexio::action]]
  void notify(name user, std::string msg) {
    require_auth(get_self());
    require_recipient(user);
  }

private:
  struct [[vexio::table]] person {
    name key;
    uint64_t age;
    std::string first_name;
    std::string last_name;
    std::string street;
    std::string city;
    std::string state;

    uint64_t primary_key() const { return key.value; }
    uint64_t get_secondary_1() const { return age;}
  };

  void send_summary(name user, std::string message) {
    action(
      permission_level{get_self(),"active"_n},
      get_self(),
      "notify"_n,
      std::make_tuple(user, name{user}.to_string() + message)
    ).send();
  };

  void increment_counter(name user, std::string type) {
    abcounter::count_action count("abcounter"_n, {get_self(), "active"_n});
    count.send(user, type);
  }

  typedef vexio::multi_index<"people"_n, person,
    indexed_by<"byage"_n, const_mem_fun<person, uint64_t,
&person::get_secondary_1>>
  > address_index;
};
```

Step 5: Recompile and redeploy the addressbook contract

Recompile the `addressbook.cpp` contract, we don't need to regenerate the ABI, because none of our changes have affected the ABI. Note here we include the abcounter contract folder with the -I option.

Shell
vexio-cpp -o addressbook.wasm addressbook.cpp -I ../abcounter/

Redeploy the contract

Shell
cleos set contract addressbook CONTRACTS_DIR/addressbook

Step 6: Test It.

Now that we have the `abcounter` deployed and `addressbook` redeployed, we're ready for some testing.

Shell
cleos push action addressbook upsert '["alice", "alice", "liddell", 19, "123 drink me way", "wonderland", "amsterdam"]' -p alice@active

Result

```
executed transaction:
cc46f20da7fc431124e418ecff90aa882d9ca017a703da78477b381a0246
eaf7  152 bytes  1493 us
#   addressbook <= addressbook::upsert
{"user":"alice","first_name":"alice","last_name":"liddell","street":"123 drink
me way","city":"wonde...
#   addressbook <= addressbook::notify          {"user":"alice","msg":"alice
successfully modified record in addressbook"}
#         alice <= addressbook::notify          {"user":"alice","msg":"alice
successfully modified record in addressbook"}
#     abcounter <= abcounter::count             {"user":"alice","type":"modify"}
```

As you can see, the counter was successfully notified. Let's check the table now.

Shell

```
cleos get table abcounter abcounter counts --lower alice --limit 1
```

Result

```
{
  "rows": [{
      "key": "alice",
      "emplaced": 1,
      "modified": 0,
      "erased": 0
    }
  ],
  "more": false
}
```

Test each of the actions and check the counter. There's already a row for alice, so upsert *should* **modify**the record.

Shell
cleos push action addressbook upsert '["alice", "alice", "liddell", 21,"1 there we go", "wonderland", "amsterdam"]' -p alice@active

Shell
executed transaction: c819ffeade670e3b44a40f09cf4462384d6359b5e44dd211f4367ac6d3ccbc70 152 bytes 909 us # addressbook <= addressbook::upsert {"user":"alice","first_name":"alice","last_name":"liddell","street":"1 coming down","city":"normalla... # addressbook <= addressbook::notify {"user":"alice","msg":"alice successfully emplaced record to addressbook"} >> Notified # alice <= addressbook::notify {"user":"alice","msg":"alice successfully emplaced record to addressbook"} # abcounter <= abcounter::count {"user":"alice","type":"emplace"} warning: transaction executed locally, but may not be confirmed by the network yet]

To erase:

Shell
cleos push action addressbook erase '["alice"]' -p alice@active

Result
executed transaction:

```
aa82577cb1efecf7f2871eac062913218385f6ab2597eaf31a4c0d25ef1bd
7df  104 bytes  973 us
#   addressbook <= addressbook::erase          {"user":"alice"}
>> Erased
#   addressbook <= addressbook::notify         {"user":"alice","msg":"alice
successfully erased record from addressbook"}
>> Notified
#         alice <= addressbook::notify         {"user":"alice","msg":"alice
successfully erased record from addressbook"}
#     abcounter <= abcounter::count            {"user":"alice","type":"erase"}
warning: transaction executed locally, but may not be confirmed by the
network yet    ]
Toaster:addressbook sandwich$
```

Next, we'll test if we can manipulate the data in `abcounter` contract by calling it directly.

Shell

```
cleos push action abcounter count '["alice", "erase"]' -p alice@active
```

Checking the table in `abcounter` we'll see the following:

Shell

```
cleos get table abcounter abcounter counts --lower alice
```

Result

```
{
  "rows": [{
      "key": "alice",
      "emplaced": 1,
      "modified": 1,
      "erased": 1
    }
```

```
  ],
  "more": false
}
```

Wonderful! Since we require_auth for `name("addressbook")`, only the `addressbook` contract can successfully execute this action, the call by alice to fudge the numbers had no affect on the table.

Extra Credit: More Verbose Receipts

The following modification sends custom receipts based on changes made, and if no changes are made during a modification, the receipt will reflect this situation.

C++

```
#include <vexio/vexio.hpp>
#include "abcounter.cpp"

using namespace vexio;

class [[vexio::contract("addressbook")]] addressbook : public
vexio::contract {

public:

  addressbook(name receiver, name code,  datastream<const char*> ds):
contract(receiver, code, ds) {}

  [[vexio::action]]
  void upsert(name user, std::string first_name, std::string last_name,
uint64_t age, std::string street, std::string city, std::string state) {
    require_auth(user);

    address_index addresses(get_first_receiver(),
get_first_receiver().value);

    auto iterator = addresses.find(user.value);
```

```
   if( iterator == addresses.end() )
   {
     addresses.emplace(user, [&]( auto& row ){
      row.key = user;
      row.first_name = first_name;
      row.last_name = last_name;
      row.age = age;
      row.street = street;
      row.city = city;
      row.state = state;
      send_summary(user, " successfully emplaced record to
addressbook");
      increment_counter(user, "emplace");
     });
   }
   else {
     std::string changes;
     addresses.modify(iterator, user, [&]( auto& row ) {

       if(row.first_name != first_name) {
         row.first_name = first_name;
         changes += "first name ";
       }

       if(row.last_name != last_name) {
         row.last_name = last_name;
         changes += "last name ";
       }

       if(row.age != age) {
         row.age = age;
         changes += "age ";
       }

       if(row.street != street) {
         row.street = street;
         changes += "street ";
       }

       if(row.city != city) {
         row.city = city;
         changes += "city ";
       }

       if(row.state != state) {
         row.state = state;
         changes += "state ";
       }
     });
```

```
      if(changes.length() > 0) {
        send_summary(user, " successfully modified record in
addressbook. Fields changed: " + changes);
        increment_counter(user, "modify");
      } else {
        send_summary(user, " called upsert, but request resulted in no
changes.");
      }
    }
  }

  [[vexio::action]]
  void erase(name user) {
    require_auth(user);
    address_index addresses(get_first_receiver(),
get_first_receiver().value);
    auto iterator = addresses.find(user.value);
    check(iterator != addresses.end(), "Record does not exist");
    addresses.erase(iterator);
    send_summary(user, " successfully erased record from
addressbook");
    increment_counter(user, "erase");
  }

  [[vexio::action]]
  void notify(name user, std::string msg) {
    require_auth(get_self());
    require_recipient(user);
  }

private:

  struct [[vexio::table]] person {
    name key;
    std::string first_name;
    std::string last_name;
    uint64_t age;
    std::string street;
    std::string city;
    std::string state;
    uint64_t primary_key() const { return key.value; }
    uint64_t get_secondary_1() const { return age;}
  };

  void send_summary(name user, std::string message) {
    action(
      permission_level{get_self(),"active"_n},
      get_self(),
```

```
    "notify"_n,
    std::make_tuple(user, name{user}.to_string() + message)
   ).send();
 };

 void increment_counter(name user, std::string type) {

   action counter = action(
     permission_level{get_self(),"active"_n},
     "abcounter"_n,
     "count"_n,
     std::make_tuple(user, type)
   );

   counter.send();
 }

 typedef vexio::multi_index<"people"_n, person, indexed_by<"byage"_n,
const_mem_fun<person, uint64_t, &person::get_secondary_1>>>
address_index;
};
```

3.8. Linking Custom Permission

Introduction

On an VEXIO blockchain, you can create various custom permissions for accounts. A custom permission can later be linked to an action of a contract. This permission system enables smart contracts to have a flexible authorization scheme.

This tutorial illustrates the creation of a custom permission, and subsequently, how to link the permission to an action. Upon completion of the steps, the contract's action will be prohibited from executing unless the

authorization of the newly linked permission is provided. This allows you to have greater granularity of control over an account and its various actions.

With great power comes great responsibility. This functionality poses some challenges to the security of your contract and its users. Ensure you understand the concepts and steps prior to putting them to use.

Step 1. Create a Custom Permission

Firstly, let's create a new permission level on the `alice` account:

Shell
cleos set account permission alice upsert YOUR_PUBLIC_KEY owner -p alice@owner

A few things to note:

1. A new permission called **upsert** was created
2. The **upsert** permission uses the development public key as the proof of authority
3. This permission was created on the `alice` account

You can also specify authorities other than a public key for this permission, for example, a set of other accounts. Check account permission for more details.

Step 2. Link Authorization to Your Custom Permission

Link the authorization to invoke the `upsert` action with the newly created permission:

Shell
cleos set action permission alice addressbook upsert upsert

In this example, we link the authorization to the `upsert` action created earlier in the addressbook contract.

Step 3. Test it

Let's try to invoke the action with an `active` permission:

Shell
cleos push action addressbook upsert '["alice", "alice", "liddel", 21, "Herengracht", "land", "dam"]' -p alice@active

You should see an error like the one below:

Text
Error 3090005: Irrelevant authority included Please remove the unnecessary authority from your action! Error Details: action declares irrelevant authority '{"actor":"alice","permission":"active"}'; minimum authority is {"actor":"alice","permission":"upsert"}

Now, try the **upsert** permission, this time, explicitly declaring the **upsert** permission we just created: (e.g. `-p alice@upsert`)

Text
cleos push action addressbook upsert '["alice", "alice", "liddel", 21, "Herengracht", "land", "dam"]' -p alice@upsert

Now it works:

Text
cleos push action addressbook upsert '["alice", "alice", "liddel", 21, "Herengracht", "land", "dam"] -p alice@upsert executed transaction: 2fe21b1a86ca2a1a72b48cee6bebce9a2c83d30b6c48b16352c70999e4c20983 144 bytes 9489 us #   addressbook <= addressbook::upsert {"user":"alice","first_name":"alice","last_name":"liddel","age":21,"street":"Herengracht","city":"land",... #   addressbook <= addressbook::notify {"user":"alice","msg":"alice successfully modified record to addressbook"} # vexio <= addressbook::notify {"user":"alice","msg":"alice successfully modified record to addressbook"} # abcounter <= abcounter::count {"user":"alice","type":"modify"}

3.9. Deferred Transactions

Previously, we sent an external inline action in the addressbook contract. In this tutorial we'll explore sending a deferred transaction, which will include the inline action we sent in the previous tutorial.

Using deferred transaction to replace inline action

Open the `addressbook.cpp` file in your favorite editor and use the following code to replace the `increment_counter` function.

C++

```
void increment_counter(name user, std::string type) {
  vexio::transaction deferred;

  deferred.actions.emplace_back(
    permission_level{get_self(),"active"_n},
    get_self(), "notify"_n,
    std::make_tuple(user, type)
  );

  deferred.send(user.value, get_self());
}
```

The code listing above does the following:

1. Initializes a transaction object.
2. Creates an action with the same parameters in the previous addressbook example.
3. Adds the action into the transaction.

4. Sends the transaction with two parameters. The first parameter is the sender id which will be useful in the later section of this tutorial. The second parameter is the account which will pay for the RAM costs associated with the deferred transaction.

Canceling a deferred transaction

To cancel a deferred transaction, call `cancel_deferred` with the sender id specified when the deferred transaction was created. In our case this is the name variable user's value.

See the example below:

C++
cancel_deferred(user.value);

Executing the deferred transaction

To test if the deferred transaction invokes the notify action, run the below command:

Shell
cleos push action addressbook upsert '["alice", "alice", "liddell", 19, "123 drink me way", "wonderland", "amsterdam"]' -p alice@active

Since the transaction was deferred, you will not see output related to the deferred transaction in the console, even though the `upsert` transaction was successfully invoked.

```
Text
executed transaction:
f1143e224c9809aafb6e7274096521168ecfe1e21feb86ca2d4794c4a4929fd5  208 bytes  428 us
#   addressbook <= addressbook::upsert
{"user":"alice","first_name":"alice","last_name":"liddell","age":19,"street":"123 drink me way","cit...
warning: transaction executed locally, but may not be confirmed by the network yet         ]
```

Although you will not see the deferred transaction output in the console, you can check the nodeos logs to see if the transaction executed successfully. To check the nodeos logs, execute the following command:

```
Shell
tail -f nodeos.log
```

```
Shell
info  2019-04-08T12:52:34.000 thread-0  producer_plugin.cpp:1596 produce_block        ] Produced block 00081d73bfb74c06... #531827 @ 2019-04-08T12:52:34.000 signed by vexio [trxs: 0, lib: 531826, confirmed: 0]

info  2019-04-08T12:52:34.502 thread-0  producer_plugin.cpp:1596 produce_block        ] Produced block 00081d7447279fcf... #531828 @ 2019-04-08T12:52:34.500 signed by vexio [trxs: 1, lib: 531827, confirmed: 0]
info  2019-04-08T12:52:35.004 thread-0  producer_plugin.cpp:1596 produce_block        ] Produced block 00081d758add241e... #531829 @ 2019-04-08T12:52:35.000 signed by vexio [trxs: 1, lib: 531828,
```

```
confirmed: 0]

info  2019-04-08T12:52:35.503 thread-0  producer_plugin.cpp:1596
produce_block        ] Produced block 00081d76113974b9... #531830 @
2019-04-08T12:52:35.500 signed by vexio [trxs: 0, lib: 531829,
confirmed: 0]
```

You can see the there are two transactions (trxs) executed consecutively. This means the deferred transaction was executed by nodeos.

Err on the side of caution

When contract code raises an exception it can be handled with error handling code. You can define your own error handling code as follows

```
void onError() {
   print("onError is called");

    }
```

With the following notify attribute:

```
[[vexio::on_notify("vexio::onerror")]]
void onError() {
   print("onError is called");

    }
```

Using the above onerror function you can handle various exceptions. This allows you to fail safely when necessary.

For more detail on how onerror works internally see here

Nondeterministic

Because you can schedule a deferred transaction at a future point in time, the **execution of a deferred transaction is indeterministic**

It may be that no nodes on an VEXIO network ever attempt to execute a scheduled deferred transaction. Since all users of an VEXIO blockchain network can schedule a deferred transaction, the to-be-executed transactions can accumulate in a backlog. It is possible that the expiration time of a deferred transaction has been reached before a deferred transaction has been scheduled to execute.

Due to this behavior and other subjective reasons you should never design your application based the deterministic execution of a deferred transaction.

For more detail on how a deferred transaction works see here

4.

SMART CONTRACT API

3.

RPC API

WALLET RPC API