

# Blockchain By Example

A beginner's guide to creating decentralized applications using  
Bitcoin, Ethereum, and Hyperledger

**EARLY ACCESS**



By Bellaj Badr

**Packt>**

[www.packt.com](http://www.packt.com)

# Table of Contents

Preface

## 1. Chapter 1 : Say Hello to blockchain

### Introduction

#### Emergence of blockchain and cryptocurrency

#### From virtual to crypto currency

#### What is a Blockchain?

#### Blocks

#### Transactions

#### Smart contracts

### Say "hello" to the Blockchain

#### Getting started

#### Run a bitcoin client for the first time

#### Synchronizing the blockchain

#### Running Bitcoin core in pruned mode

#### Run Electrum client:

### Method 1: Building raw transaction using bitcoin client

#### Funding our addresses

#### Unspent Transaction Output

#### Creating the transaction

## Transaction structure

Signing the transaction

Sending the transaction

Retrieve your message online from the blockchain

Retrieve your message from the local blockchain

Method 2 : Build a raw bitcoin transactions in JavaScript .

Preparation

Let's code

Types of blockchains

Classification of blockchains

Summary

## 2. Chapter 2 : Building a Bitcoin payment system

Introduction

What is Bitcoin?

Why choose Bitcoin as platform?

Getting started with Bitcoin

&#xA0;Setting up regtest environment

First commands

Transactions in Bitcoin

Mining&#xA0;

Bitcoin Scripting

Building p2pkh script using JavaScript

Building a custom script

Building a payment gateway

Project description

Bitcoin payment protocol overview

Prerequisites

Project setup

Certificate management

Merchant side

Building payment request URI

Routing

Checkout view

Proceeding with payment

Requesting payment details

Receiving and acknowledging payment

Invoicing

Client side

Preview your application

Payment protocol using BITCOINJ

Prerequisites

BitcoinJ installation

BitcoinJ client

Bitcoin wallet configuration

Requesting payment

Sending a payment

Testing code

Bitcoin smart contracts

What's Rootstock

Rootsock setup

Interactions with RSK

Accounts settings

Writing your first Bitcoin smart contract

Deploying the contract

Interacting with the contract

Summary

### 3. Chapter 3 : Building your own cryptocurrency

Introduction

Compiling Bitcoin from source

Preparing your build system

Installing dependencies

Bitcoin-qt: Qt5 GUI for Bitcoin

Cloning the Bitcoin source code

Building Bitcoin Core

Checking your build worked

New cryptocurrency: ReaderCoin

Cloning Bitcoin

Readercoin rebranding

Changing the ports

Changing pchMessageStart

Genesis block

New pszTimestamp

New nonce, epoch time, and nbits



New genesis hash

Editing the Merkle root

Removing the seed nodes

Checkpoints

ChainTxData

Rewarding

Halving

Total supply

POW parameters: target

New block time

Difficulty adjustment time interval

Difficulty adjustment rate

Time maturity

Block size

BIPs: Bitcoin Improvement Proposals

Compiling and testing

Design and graphics (Qt wallet)

Redesign of the Bitcoin-Qt wallet

Graphical interface addition

Building Readercoin network

Readercoin mining

Sending transactions

Building a transaction and block explorer

Iquidus setup requirements

Creating Iquidus' database

Installing Iquidus

Iquidus Configuration

Syncing databases with the blockchain

Wallet preparation

Starting the explorer

Building an exchange market

Summary

#### 4. Chapter 4 : Peer to peer auction in Ethereum

## 5. Chapter 5 : Tontine game with Truffle and Drizzle

## 6. Chapter 6 : Blockchain-based Futures system

## 7. Chapter 7: Blockchains in business

## 8. Chapter 8: Creating an ICO

## 9. Chapter 9: Distributed Storage: IPFS and Swarm



## 10. chapter 10 Supply chain on Hyperledger

## 11. Chapter 11 Letter of credit (LC) Hyperledger

# Preface

Chapter 1, *Say Hello to Blockchain*, the purpose of this chapter is to introduce several elementary notions and to give an outline of the background information regarding the development of blockchain. The reader will get its first experience with the blockchain by a simple data exchange with the blockchain network.

Chapter 2, *Building a Bitcoin Payment System*, Bitcoin is the key point in the blockchain space. In this chapter, the reader will get started with the Bitcoin project by building a payment application using Bitcoin as development platform.

Chapter 3, *Building Your Own Cryptocurrency*, the reader will acquire solid knowledge about the Bitcoin code base and he will be able to build its own currency.

Chapter 4, *Peer to Peer Auction in Ethereum*, Ethereum is the second largest blockchain project after Bitcoin. This chapter will provide a gentle introduction the Ethereum project and presents all its features by building a peer to peer lending platform.

Chapter 5, *Tontine Game with Truffle and Drizzle*, we will cover in this chapter advanced Ethereum features and solidity functionalities by proposing to build a Tontine Bank.

Chapter 6, *Blockchain-Based Futures System*, this chapter focus on the integration of the blockchain technologies with the existing platforms and applications, it helps the reader to understand how he can integrate the blockchain in his business and applications.

Chapter 7, *Blockchains in Business*, this chapter will teach the reader how to use Ethereum as a private chain solution for internal business or in B2B.

Chapter 8, *Creating an ICO*, via this chapter the readers will be introduced to the ERC20 smart contract standard. It constitutes a complete guide to setup an Ethereum Token based project.

Chapter 9, *Distributed Storage: IPFS and Swarm*, as the storage an important layer for the applications, this chapter tends to present with examples the most important storage and file system infrastructures based on the blockchain.

Chapter 10, *Supply Chain on Hyperledger*, Hyperledger is one of the biggest project in the blockchain industry. It is a global collaboration, hosted by The Linux Foundation, including leaders in different sectors. This chapter presents an introduction to on Hyperledger by explaining how create a supply chain application.

Chapter 11, *Letter of Credit (LC) Hyperledger*, this chapter includes a breakdown of the essential components of Hyperledger project. It presents also to the reader the occasion to develop and manage scalable, highly interoperable business solutions based on Hyperledger.

# **Chapter 1 : Say Hello to blockchain**

# Introduction

What is blockchain? Certainly, with the huge hype around, you must have heard or come across this question - it might be even the reason why you are reading this book. Let's discover in this first chapter from a developer's standpoint, what's behind the Hype.

As you might know, the Blockchain is an emerging technology that has the potential to dramatically revolutionize many different fields. This potential is primarily based on its ability to offer people a trustworthy channel to transfer value or real assets (Tokenization) over internet. Moving us thus, from the Internet of Information to the Internet of value and potentially causing breaking changes to the existing financial systems.

Blockchain is for good reasons a revolution similar to internet, not a passing trend as it presents a solution for an old unsolved financial dilemma. Indeed, for first time in history we are able to establish the trust within trust-less environment (such Internet) without relying on a trusted authority. This ability earned blockchain the title of Trust Machine.

The potential impact of the blockchain is huge, it goes far beyond the mere decentralization of the financial sector. In fact, its ability to short-circuit intermediaries opens the door to redefine almost every field revolving around technology even internet pushing us toward a peer to peer world.

Through this short introduction, I am trying to give you a foretasting of the importance of our topic and to confirm your timely choice of learning about such revolutionary technology. As the book's name

suggests, the approach we will be following throughout this book is to build concrete blockchain projects instead of laying out abstract concepts. Nonetheless, in spite of its less technical nature, the prime objective of this introductory chapter is to provide you with the needed background to build the various projects presented in this book.

In this chapter we will cover the following topics :

- What is Cryptocurrency?
- What is blockchain?
- How to send and receive Bitcoins?
- How to store data into the Bitcoin's blockchain using JavaScript?
- Overview of Blockchain types

However, this chapter doesn't intend to cover:

- Blockchain's underlying cryptography.
- Cryptocurrency trading.

In this chapter, the first part will be dedicated to introduce basic concepts, and the second part will be practical, hence we will discover how to interact with the Bitcoin Blockchain by using the famous “Hello World” example to get you started.

# Emergence of blockchain and cryptocurrency

Many find it hard to understand the logic and the concepts behind the blockchain or why would they need it. This misunderstanding happens primarily when we don't have an idea about what problem it solves, or what advantages it promises. Therefore, I believe it would be necessary to clarify at the first stage the problem solved by the blockchain. This attempt will guide us to start by learning about the cryptocurrency concept and history.



# From virtual to crypto currency

The blockchain didn't appear out of blue but it was the outcome of the evolution of Fintech and virtual currencies that has taken place over decades. At the end of the last century, the widespread use of the internet favored the emergence of digital currencies as extension of the electronic cash systems. Many projects were developed to create new digital currencies, to name a few: E-cash, E-gold, Webmoney, Liberty reserve, etc.

After a huge success in the 90s, these projects had ceased to exist by the beginning of the new century whether by going bankrupt or being **stopped by authorities**. A currency, which could disappear overnight, is a real nightmare and with the digital currency, this situation was inevitable due to the centralized nature of such systems. There was always a need for a central authority to fight frauds and manage trust within the system. To avoid this fatal feebleness, the opposite decentralized model was presented as a solution, however it was hard to establish the trust in such environments without any central authority. This contrast makes creating a digital, unstoppable and reliable currency a disentangled Gordian knot. Thankfully, the progress of cryptography and the emergence of some clever solutions like proof of work (Hashcash Project) gave hope to break the deadlock .

In fact, in 2008 a mysterious person called Satoshi Nakamoto, responded to that challenge and unveiled a digital currency called Bitcoin. This new currency effectively harnessed cryptography techniques to manage ownership and to secure the system, hence the name crypto-currency instead of virtual currency. Satoshi solved in a

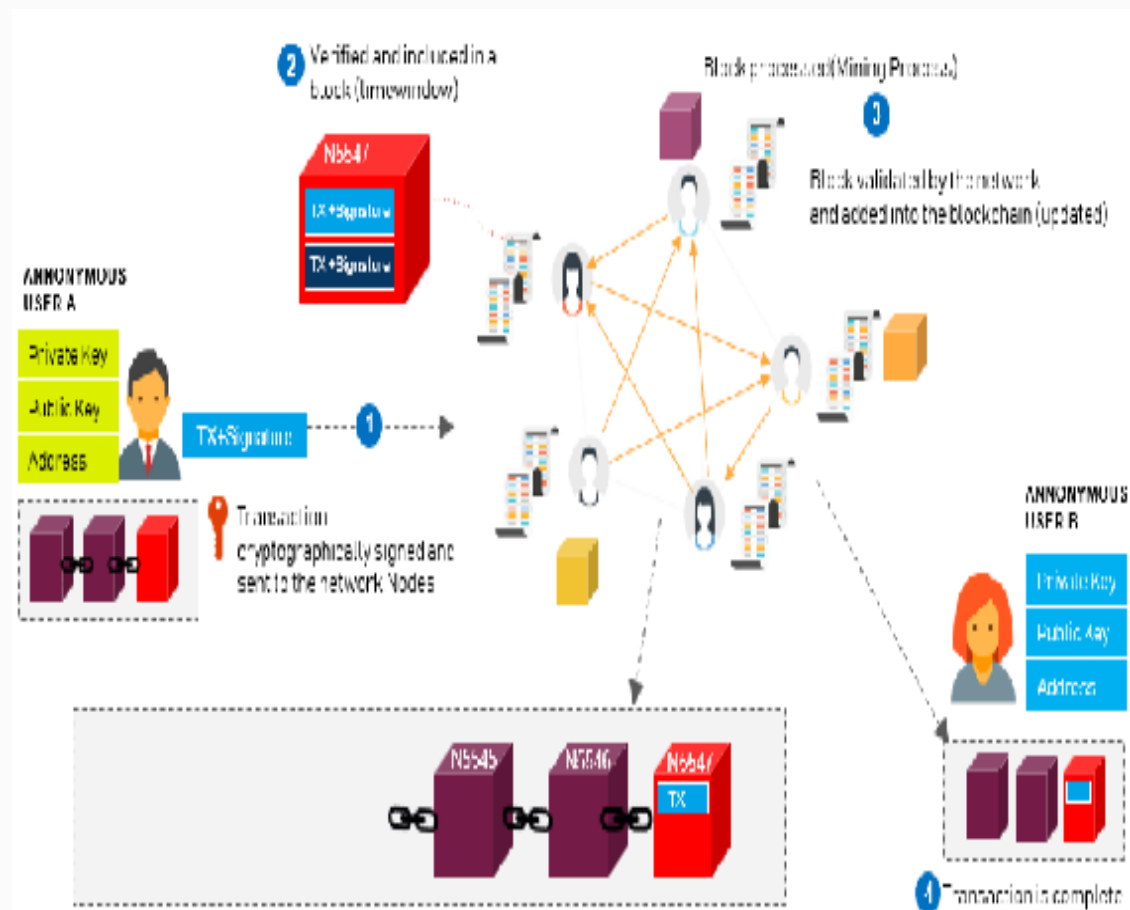
clever way, the aforementioned problems by introducing what he called initially "a chain of blocks". In his published white paper, he presented his vision for a new Peer-to-Peer electronic cash system -- Bitcoin and defined in details the mechanics of its underlying machinery--blockchain. Bitcoin, was the first reliable and distributed electronic cash system that's fully peer-to-peer, underpinned by the following basics:

- Encryption to ensure ownership and identity.
- Proof-of-work consensus mechanism for validating transactions and securing the network against double spending.
- Transparent and shared ledger (Blockchain).
- Pseudonymity.

With the assumption that the network majority (>51%) is honest, the bitcoin system operates autonomously following the rules defined by the protocol(consensus rules) to validate a given transaction. By using the shared blockchain, each player has the ability to check individually the transactions log history and the sender's solvency, then votes whether the proceeded transaction is valid or not. The voting depends on the overall hash-power the player puts in service to secure the network (initially 1 CPU is 1 vote).

Practically, to use a cryptocurrency users should install a specific client which creates for them a wallet, generates cryptographic key pairs (private/public keys), and sync the blockchain with the network. The public key is used by the client(software) to generate valid addresses and the funds sent to a given address are controlled by the private key from which the address was calculated, thus we rely on secure cryptographic principles to manage ownership.

The following illustration depicts how transactions are processed in bitcoin's peer-to-peer network and added into the blockchain:

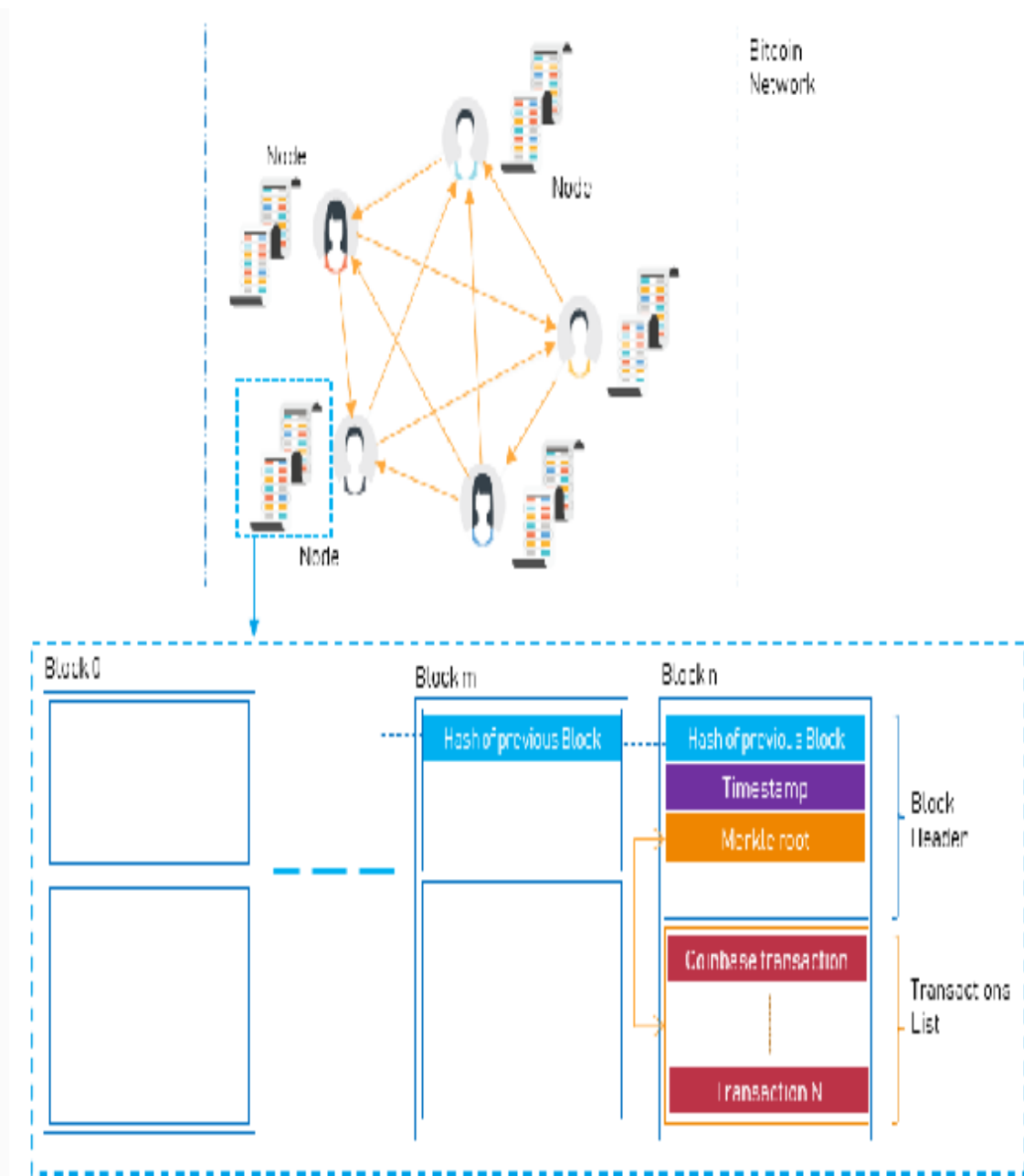


For bitcoin network where users don't know each other, the blockchain is considered as the single source of truth to which they refer to learn about the previous consensus outcomes. The blockchain with the consensus protocol permitted that the network operates without a single point of failure managing the transactions.

# What is a Blockchain?

Often mistakenly confused with bitcoin, the blockchain is the underlying technology used by bitcoin to operate. Concretely, it's an append-only and chronologically (timestamped) growing database which harnesses basic cryptographic measures to protect stored transactions from tampering (data can't be deleted or altered).

This database or ledger collects and records monetary transactions validated by the network in elementary units called Blocks. Once validated by the network consensus mechanism, these blocks are added to an existing sequential chain of cryptographic hash-linked blocks to ensure the integrity of the data, hence the name blockchain. Consequently, if a single bit changes in a block the hash-link collapses and therefore the chain is broken and it will be rejected by the network.



As presented in the previous figure, the blockchain is replicated, and processed by the members of the network to ensure that everyone has a consistent view of the transaction log and when a new block is validated all nodes synchronize the same copy. It shows us also that the blockchain implements a special data structure that consists of linked blocks storing transactions and smart contracts. Let us take a closer look at these key elements in detail.



# Blocks

If we consider the blockchain as a ledger or a book, a block would be perceived as a page or a table in which we record a collection of confirmed transactions. Each block stored in the blockchain is uniquely identified by a hash and composed of a header and a body. The header encloses information about its creation (Timestamp, Merkle root, Nonce, Difficulty Target, version) and a reference to a previous block, whereas the body is a collection of the accepted transactions. When a block is successfully validated (mined) it becomes part of the official block chain and new bitcoins are generated in the block (Coinbase transaction) and paid to the validators (aka miners).

# Transactions

Transactions are the most fundamental building blocks of the blockchain system. They represent transfer of value (cryptocurrency) within the blockchain network between two addresses. More tangibly, they are represented by small data structures defined by the blockchain protocol (e.g. Bitcoin, Ethereum, ...) which specifies their attributes (Metadata, inputs, outputs, ...) and model.

Before broadcasting the transaction, the user sending the funds signs it using his private key (managed by his wallet) and specifies the destination address. Digital signatures and public key are used to enable anyone to validate the transaction and to check if the sender has the right to spend the bitcoins held by a specific address.



# Smart contracts

Smart contracts are one of the most exciting concepts in the blockchain. They represent self-executing scripts stored on the blockchain. The smart contract takes the blockchain to the next stage enabling it to translate business logic into inviolable contract terms, which will be autonomously executed without relying on a broker, lawyer or other intermediaries. The early form of smart contract was defined in bitcoin using basic scripts (locking and unlocking scripts), but the concept evolved with the emergence of other blockchains.

Smart contracts are one of the powerful disruptive forces of the blockchain which is garnering more and more business attention as witnessed by the Gartner report. They estimate that by 2022, smart contracts will be in use by more than 25% of global organizations. Aware of their importance, we will reserve an important part of this book to introduce you to smart contracts in leading blockchain platforms --Bitcoin, Ethereum and Hyperledger.

We are done with the concepts let's practice a little bit to understand what was presented so far.

# Say "hello" to the Blockchain

The blockchain as a technology evolves rapidly with emergence of new techniques deriving from the proliferation of blockchain projects. Hence the attempts to understand the present day Blockchain machinery more closely leads to the discovery of Bitcoin. Therefore, in this chapter we will adopt bitcoin as the main example. This choice is supported by the fact that bitcoin is the genesis blockchain implementation and almost all the other projects mimic its design and mechanics.

In the following sections, we will connect to the bitcoin network and store the classic "hello world" message into the blockchain. Bitcoin transactions can be used to store small amounts of data in the blockchain - allowing developers to build distributed systems on top of Bitcoin, such as `colored coin`, `Counterparty`, `Tieron`, etc. By exploring the Bitcoin blockchain, you would be surprised by the amount of hidden messages stored in the bitcoin blockchain.

# Getting started

In order to store our message into the blockchain, we will setup two bitcoin clients (receiver and sender), then we will build a raw transaction sending 1 Bitcoin along with our message.

Technically speaking, one of the best-known practices for storing data in the bitcoin blockchain is to create a zero-value `OP_RETURN` output. As defined in bitcoin's protocol, the `OP_RETURN` script opcode enables us to store up to 80 bytes. You can check it out in bitcoin's code base (`script/standard.h`):

```
static const unsigned int MAX_OP_RETURN_RELAY = 83;
```

As commented in the header file `standard.h`, the 3 additional bytes are for the needed opcodes and the remainder is for the extra message.

More importantly, `OP_RETURN` output can be pruned and helps to avoid bloating the blockchain in the future. Don't worry if you feel lost, we will dive deep into bitcoin's concepts such as outputs and scripting in the next chapter.

To reach our goal, I am planning to use two different methods :

1. Create a raw transaction with an `OP_RETURN` output using RPC commands and a bitcoin client.

2. Write a NodeJs program to create and send the raw transaction using an online REST API.

The second step will require some familiarity with the JavaScript programming language.

# Run a bitcoin client for the first time

A bitcoin client is the end-user software that facilitates to perform bitcoin operations (sending transactions, receiving payments, etc.) when you run one you become part of bitcoin network. Among many, we choose for this guide two common clients: the Bitcoin Core client and Electrum. In this scenario, the sender will use the Electrum client whereas the receiver will be using the bitcoin core client (the most popular bitcoin client). For the purpose of this manipulation, I will demonstrate installing them on a single machine using Ubuntu 16.04.

You can install Bitcoin core client by following the commands below (further instructions are available at <https://bitcoin.org>):

```
sudo add-apt-repository ppa:bitcoin/bitcoin
sudo apt-get update
sudo apt-get install bitcoind bitcoin-qt
```

Electrum is a lightweight wallet which means it doesn't require downloading the whole Blockchain as we will see in the next section. You can download and install the latest version of Electrum as follows:

```
wget https://download.electrum.org/3.2.2/Electrum-3.2.2.tar.gz
sudo apt-get install python3-setuptools python3-pyqt5 python3-pip
sudo pip3 install Electrum-3.2.2.tar.gz
```

Once both clients are installed we need to synchronize them with the network.



# Synchronizing the blockchain

We have learned earlier that the blockchain is a transaction database duplicated by all computers on the network. Practically, to use bitcoin properly we have to sync a voluminous amount (>200Go) of data to enable sending or receiving bitcoins. However, there are two workarounds to overcome this situation:



1. Enabling pruned mode for a full node client like bitcoin core.

2. Using a thin (SPV) client like Electrum which fetches blockchain information from Electrum servers instead having a local copy.

In this guide we will look at both solutions. Nonetheless, it's always advisable to use your own Bitcoin full node to benefit from the power of the blockchain.

# Running Bitcoin core in pruned mode

Depending on your OS you need to create the configuration file `bitcoin.conf` in the default data directory located under the following paths :

- **Windows:** `%APPDATA%\Bitcoin\`
- **Mac:** `$HOME/Library/Application Support/Bitcoin/`
- **Linux:** `$HOME/.bitcoin/`

For Linux create a `.bitcoin` directory using `mkdir ~/.bitcoin` then create `bitcoin.conf` file using : `nano ~/.bitcoin/bitcoin.conf`

Then add the following lines to your `bitcoin.conf` to define your client configuration (The comments after '#' sign introduce each parameter).

```
rpcuser=user_name           #Username for JSON-RPC connections
rpcpassword=your_password   #Password Username for JSON-RPC
connections
server=1                     #Tells Bitcoin-Qt and bitcoind to accept
JSON-RPC commands
testnet=1                    #Run on the test network instead of the
real bitcoin network.
prune=550                     #Enables pruning mode
```

Once copied press `CTRL+X` , `Y` then `Enter` to save the file.

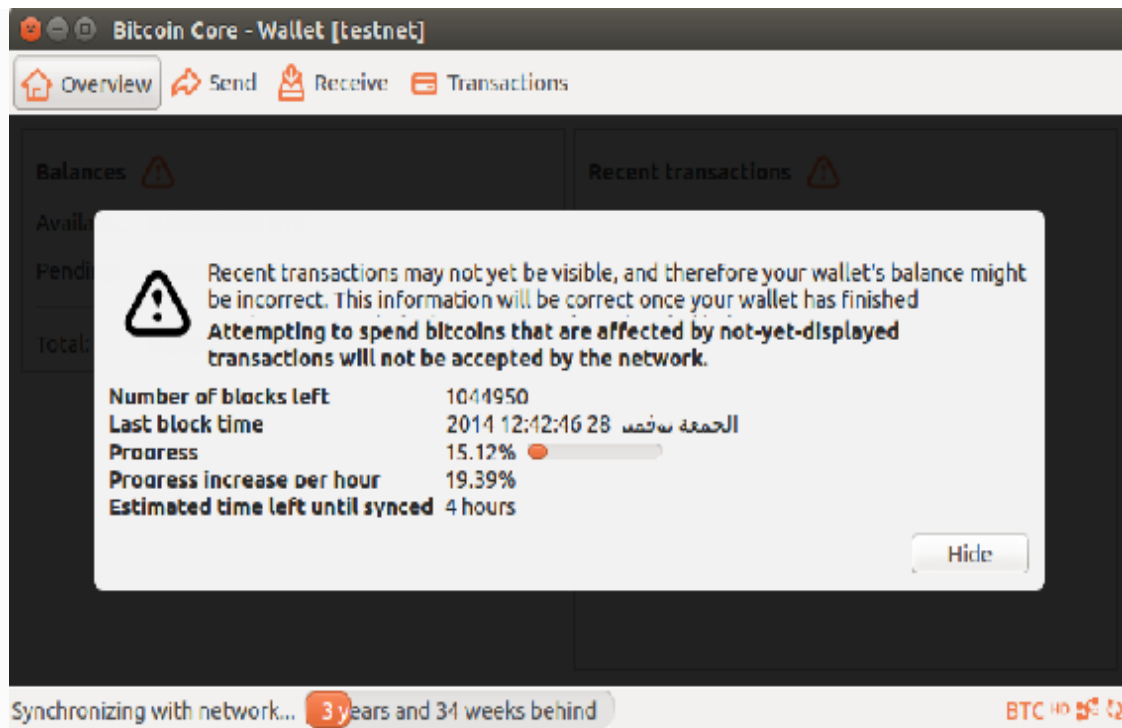
Now our first client is ready to run on the `Testnet` which is a bitcoin network created for testing purpose that follows the same rules as main network. It's a public network using worthless bitcoins, therefore you can use this network to send for free your transactions and test your applications.

*At the time of writing, the volume of the entire blockchain exceeds 200 GB. Therefore, we activate pruning mode by setting the parameter `prune=<n>` in `bitcoin.conf` such that "`n`" indicates the space you are willing to allocate to the blockchain in MiB with a minimum of 550 Mib. Note that the data directory will exceed a few GB (2 GB in my case) because it contains more than just the blockchain. It hosts additional index and log files along with the UTXO database. The prune size only defines how many blocks to be downloaded.*

It's now time for running the bitcoin client. Open a new CLI (command-line user interface) window and run the following command:

```
bitcoin-qt
```

As a result you will get bitcoin running with its standard GUI interface and connected to the testnet network. For the first run, it will ask you to set the data directory, which we should set to the default. Afterwards, it will automatically create a wallet for you and start syncing with the testnet network and download the blockchain.

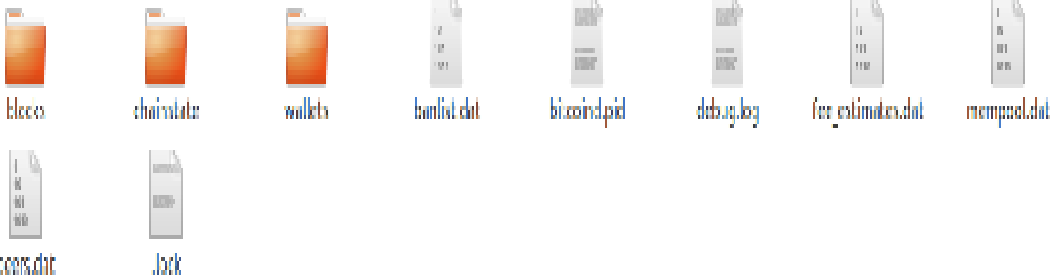


Alternatively, you could run the bitcoin daemon in CLI mode by running the following command:

```
bitcoind
```

It's up to you to choose which mode to continue using (bitcoind or bitcoin-qt), the available RPC commands are the same. For my part, I'll continue this guide using bitcoin-qt.

As the Bitcoin core client starts, it creates many sub-directories and files in the default Data Directory ( .bitcoin) as shown in the following picture:



The main sub directories are :

- **blocks** : Stores actual Bitcoin blocks.
- **chainstate**: Holds LevelDB database for available UTXOs (an abbreviation of Unspent Transaction Output) or in other terms a database storing how much money everyone has
- **wallet**: Contains encrypted `wallet.dat` file which stores the private keys.

Even if the network sync is not finished yet, you can open the `blocks` sub-directory to visualize the blockchain's blocks stored in raw format. Practically, each `blk00*.dat` file is a collection of several raw blocks. We will read later the content of one of these files.



More details about the content of `.bitcoin` directory can be found in the official documentation : [https://en.bitcoin.it/wiki/Data\\_directory](https://en.bitcoin.it/wiki/Data_directory)

While the server (bitcoind or bitcoin-qt) is running, open another terminal and let's generate a new address for our wallet by executing the command `bitcoin-cli getnewaddress`, as in the following picture:

```
user@ByExample-node:~$ bitcoin-cli getnewaddress  
2MsHsi4CHXsaNZSq5krnrpP4WShNgtuRa9U
```

`bitcoin-cli` is a tool that enables us to issue RPC commands to `bitcoind` or `bitcoin-qt` from the command line (`bitcoin-qt` users can access the bitcoin RPC interface by using the Debug console, under the Help menu).

Now we are done with `bitcoin-core`, let it sync the blockchain copy and move on to configure Electrum.

# Run Electrum client:

After you have downloaded and installed Electrum, open Electrum's testnet mode by running : `electrum --testnet` . When you run Electrum the first time it will display the new wallet creation wizard follow then these steps:

- Select in the first dialogue box “Auto Connect” and click “Next”.
- Select “Standard wallet” and click “Next”.
- Keep pressing "Next" for the rest of the dialog boxes that appear until you are requested to save the seed words. Copy them somewhere, then in the next Dialogue Box write them correctly in the given order .





- In the last step it will ask you for a password, which you can leave empty this time.

Once finished, Electrum will generate for you a new wallet with plenty of new addresses. Quit the Electrum GUI and let's continue in CLI mode. We run Electrum as a daemon process whereby we execute the JSON/RPC commands as following :

```
electrum --testnet daemon
electrum --testnet daemon load_wallet
```

In a new terminal window run `electrum --testnet listaddresses:`

```
user@ByExample-node:~$ electrum --testnet listaddresses  
[ecc] warning: libsecp256k1 library not available, falling back to software implementation.  
  
"n3CKupfRCJ6Bnmr78mw9eyesZUskfYHcPy",  
"mhc5YipxN6GhRRXtgakRBjrNUCbz6ypg66",  
"msvrqSHWjFFhS17ReQjMJyd1y4S6n7hirM",  
"mjSiYuBwjS8qeTmK44x97jG7Zmfq3ZB9Ai",  
"mq8izx8q2ydz6P3jhrPjqapMSurkkwzaTD",  
"myE8qY9MDkRxsuFPUHVZrueMPZxc9TcZBU",  
"mqWMijZC7fdFDWao2KpcGWHgnEviqNN3xu",  
"mwmMHzcxBcucgMG1E4CmXK2JJGMwbvEF9Mt",
```

At this level we have the necessary playground to start transacting with Bitcoin network.

# Method 1: Building raw transaction using bitcoin client

For sake of brevity, we'll focus herein on the instructions needed to create and send raw transactions in Bitcoin in detriment of extended explanations. In spite of that, don't worry if you don't understand all of what you read right away, in the next chapter we will present the new concepts introduced in this section (inputs, outputs, scripts...).

# Funding our addresses

First off, we need to fund our previously created address with some bitcoins to make the first transaction. Thankfully, in the testnet we can use a free funding source called a "bitcoin faucet" which provides worthless bitcoins used to test applications. For this example, browse to the online faucet website <https://testnet.manu.backend.hamburg/faucet>, and get a few by providing the first address generated by Electrum and the address created by bitcoin core as shown in the following picture:

### Faucet

I'm happy to serve you with TestNet Bitcoins. So take as much as you need to build awesome and secure BTC services. Really, just take them. You need a certain (huge) amount of TestNet Bitcoins? Contact me

You can get bonus coins:

- 110% by using HTTP?
- 120% by using a non-windows operating system
- 170% by using IPv6

Current wallet's balance is **0.4588517**. You'll get **0.5** (+ **0.05** bonus)

**Send TX ID:** 0791521852706795583caadB08006cd58bed6d7bce59cd6ca2065cd9b1070

(Or scan QR code with your camera.)

# Unspent Transaction Output

After sending the bitcoins from the faucet, let's check if the bitcoin client can see the transaction. For that, we need to list the available UTXO in both clients using the `listunspent` RPC command. Starting with bitcoin core, in your terminal window run:

```
bitcoin-cli listunspent
```

Which results in :

```
[{ }]
```

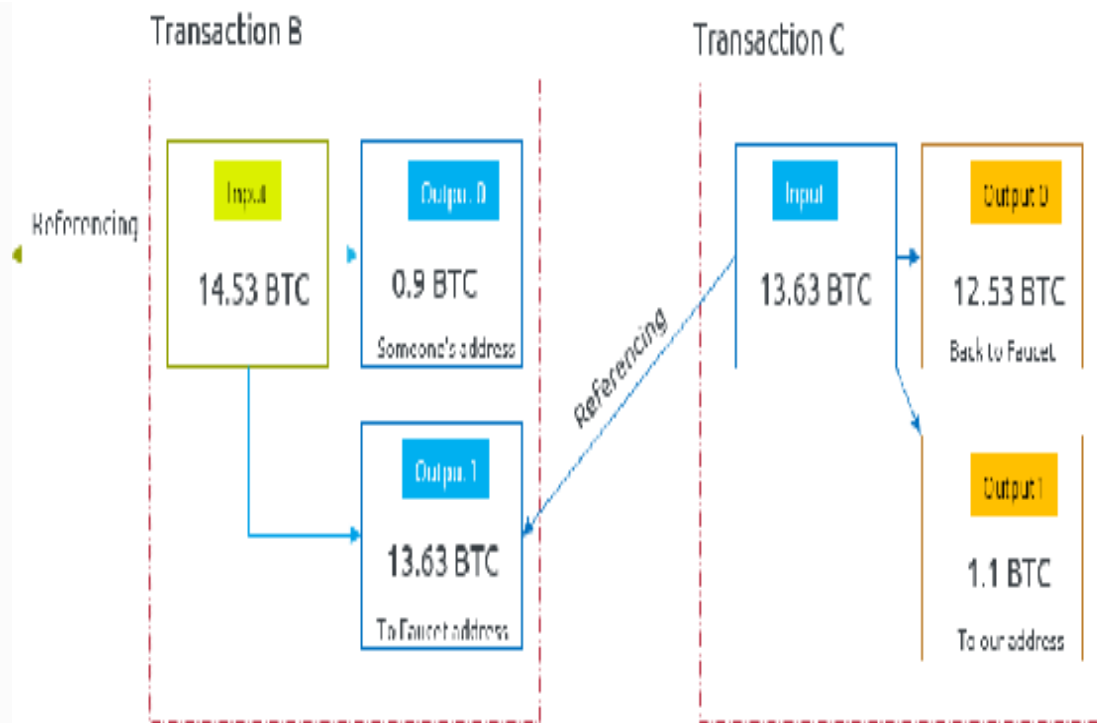
`listunspent` returns an empty result because your bitcoin core client hasn't finished syncing the blockchain, which takes time (a few hours). For this reason we will go with Electrum over Bitcoin Core for the remainder of this guide as it avoids us waiting hours to visualize the received bitcoins (unspent outputs). However, we will keep using bitcoin-core from time to time as it has a powerful command line to deal with raw transactions. If we run the same command :

```
electrum --testnet listunspent
```

we will get a list of available entries such as :

```
user@ByExample-node:~$ electrum --testnet listunspent
[ecc] warning: libsecp256k1 library not available, falling back to python-eccdsa
[
  {
    "address": "n3CKupfRCJ6Dnmr78mw9eyeszUSkfylcPy",
    "coinbase": false,
    "height": 1356645,
    "prevout_hash": "0791521362520725683caedf9980066cf68b1cd017be1694ef0daca265d9b4252",
    "prevout_n": 1,
    "value": "1.1"
  }
]
```

The previous command's output shows that we have a single available transaction uniquely identified by its hash (prevout\_hash field) received from the faucet with 1.1 bitcoin. More precisely, we have an available unspent transaction output from a previous transaction, which can be used as inputs for the transaction we are willing to build.



In bitcoin, transactions are special as they spend outputs from prior transactions and generates new outputs that can be spent by transactions in the future. In fact, users can move funds solely by spending unspent transaction outputs which are usually referred to as "UTXO"s.

The previous diagram shows the transaction we received from the faucet consumes as input an output from an old transaction received by one of its address and creates two outputs : one for us and the other returns back the change. The reason for that is transaction outputs must be fully spent.

Unlike what you might have expected, in Bitcoin transactions don't update a global user balance (Account/Balance Model) but they move bitcoins between one or more inputs and outputs (UTXO model). The total balance is calculated by the bitcoin client as the sum of the values transferred by the received unspent transactions.

## Creating the transaction

At this level, it's time to create a transaction that spends the received transaction. From the `listunspent` output we have the needed ingredients (`prevout_hash` and `prevout_n`) to construct our raw transaction. Let's see how.

First, you need to convert the message "hello world" into hexadecimal, using an [online converter](#). The hexadecimal encoded form will be the value :  
68656c6c6f20776f726c64.

Then we have to use `createrawtransaction` command which creates a transaction spending the given inputs and creating new outputs. We have to pass as arguments (from the previous output) an object with the following parameters :



1. "txid" of one of the available outputs

2. The index vout (prevout\_n for electrum) of selected output.

3. The hexadecimal form of the message

4. The destination address (created earlier)

5. The total amount of satoshis (the smallest unit of the bitcoin currency ) to send.  
we send here 1 BTC, you can set it to 0.

```
bitcoin-cli createrawtransaction "[{"txid\":\"0791521362528725683caedf998006cf68b1cd817be1694ef0daca265d9b4252\", \"vout\": 1}]" [{"data\":\"68656c6c6f20776f726c64\", \"2MsHsi4CHXsaNZSq5krnrpP4WShNgtuRa9U\":1.00000000}]"
```

You'll get the following serialized long hex-encoded string representing our raw transaction:

```
020000000152429b5d26cadaf04e69e17b81cdb168cf068099dfae3c6825875262135291070100000000ffffffffff0200000000000000000000d6a0b68656c6
```

*To facilitate the usage of the previous CLI commands (avoid manipulating long hex strings) you can assign the command `createrawtransaction` output to a terminal variable and use this later as argument for the other commands. For example we can use `RAW=$(bitcoin-cli createrawtransaction .....)` then the resulting hexadecimal string will be stored in "RAW" variable and accessible using `$RAW`.*

# Transaction structure

At first sight the previous resultant hexadecimal string seems ambiguous and meaningless. The following table breaks down and examines in depth our transaction byte per byte:

| REPRESENTATION  |   | LENGTH<br>(Bytes) | VALUE   |
|---|---|-------------------|---|
| version   |   | 4                 | 02000000  |
| input count   |   | 1                 | 01  |
| Input 1   | previous output ID in little endian (reversed)                        | 32                | 52429b5d26cadeaf04e69e17b81cbb168cf06809<br>9d1ac3c582b87b2e213b29707 |
|   | Vout (previous output index)  | 04                | 10000000  |
|   | ScriptSig length  | 01                | 00  |
|   | ScriptSig   |                   |   |
|   | sequence  | 04                | ffffff  |
| Number of generated outputs   |   | 1                 | 02  |
| Output 1  | amount of Satoshis (1 Satoshi = 0.00000001 Bitcoin)                   | 8                 | 0000000000000000  |
|   | script length   | 1                 | 0d  |
|   | ScriptPubKey:<br>OP_RETURN «Push size» «Payload (hello world in hex)» | 13                | 6a<br>0b<br>68656c6c6f70776f727664                                    |
|   |   |                   |   |
| Output 2  | Value in Satoshis (hex encoded in little endian order)                | 8                 | 00e1f50500000000  |
|   | Script length   | 1                 | 17  |
|   | ScriptPubKey (Actual output script)                                   | 23                | a914cb949dbde8d3b775bc713c20b6a62c711403<br>bd90e7                    |
| Locktime (the earliest time a transaction can be added to the blockchain) |   | 04                | 00000000  |

As you can see, our transaction has one input (the only unspent transaction received from the faucet), with transaction id

"0791...252", and two outputs:

- `OP_RETURN` output with an `OP_RETURN` script.
- An output sending 1 BTC to the specified address.

The transaction structure can be visualized by decoding back the raw transaction using the `deserialize` command. If you run `electrum --testnet deserialize <Raw transactions>`, it will output a meaningful JSON representation of our constructed transaction :





# Signing the transaction

At this point, the transaction is created but not yet transmitted to the network. To send our transaction we need to sign it using the command `bitcoin-cli signrawtransaction`. We sign the transaction using our private key (related to the receiving address) to prove to the network our ownership of the output and therefore spend the held bitcoins.

The first step will be to extract the private key associated with the first address used to receive the bitcoins from the faucet:

```
electrum --testnet listaddresses | electrum --testnet getprivatekeys -
```

Notice the presence of a dash at the end of the command. It will be replaced by the values returned from the pipe. As a result, you'll get a list of private keys. Copy the first one without the `p2pkh` prefix.

```
user@ByExample-node:~$ electrum --testnet listaddresses | electrum --testnet getprivatekeys -
WARNING: ALL your private keys are secret.
Exposing a single private key can compromise your entire wallet!
In particular, DO NOT use 'redeem private key' services proposed by third parties.
[
  "p2pkh:c0x4Ucd3uXCoe3BNns1JJ04oHn5d1chfchfIISKRaDNZOYA1FYnr",
  "p2pkh:cVZn6974DwrPHVPCK5z63VfPebk3HVqbgCPpDTEd3nw2KRUuSJE",
  "p2pkh:cSKqHn1to7E1bNCmA7QW9FelmXBKZ1HwgMaRH8YocJ44ghz1oYNg",
  "p2pkh:cQ+uxDStlrgVczKKIjMhKnSsvsowVFUJwUfowckhd9HWR+PSuKuu",
  "p2pkh:cQl23Vcj4dXxcn4ZVc1LrrvziFiJb8Avx4b5GnnSy6cMtVCYcv15",
  "p2pkh:cRHUC8nUf5fTWGyq1IIdetbqcvTdDzhskTtJkjmxeVjrqrJxAzNW",
  "p2pkh:cRkLN73Hvu7EvPNxUQTxZ6733h9wiQ91FcbuCovihKcn3RD5GV3R",
  "p2pkh:cSU4oqxnlw3TwsA8kzRh5zjQuUYErLOFpHYnEKkoLSDjb8S1NH4L"
]
```

*Beware, you should not share your private keys in real life. Doing so can lead to loss of money, as whoever has the private key can spend the received bitcoins.*

Next, we need to get the `scriptpubkey` from the output we are willing to spend. For that, firstly we have to retrieve the transaction from the blockchain using :

```
electrum gettransaction --testnet
"0791521362528725683caedf998006cf68b1cd817be1694ef0daca265d9b4252"
```

Secondly, we use the resultant raw form to get the `scriptpubkey` using :

```
electrum deserialize --testnet
0200000001915bf222c2e4e6ff36760168904ae102a0e968d83b3c575077d5475aa94dd9bf010000006b483045022100b129bc0fb5631aa668c48bb7a8f
```

Unlike before, we are here loading and deserializing the received transaction from the faucet. We will get the outputs created in this transaction :

```
"outputs": [
  {
    "address": "nxPd383YeM6tw3cCqfrmrdcXKG6PLN2f8j",
    "prevout_n": 0,
    "scriptPubKey": "76a914b0172e192d2895ea52fa975847eca0657c38f6f888ac",
    "type": 0,
    "value": 1253185750
  },
  {
    "address": "n3CKupfRCj6Bnmr7Bmw0oyeszU5ktyHcPy",
    "prevout_n": 1,
    "scriptPubKey": "76a914edccc09f510bf958a6acc8a79cb28a745c039e27088ac",
    "type": 0,
    "value": 110000000
  }
],
"partial": false,
"segwit_per": false,
"version": 2
}
```

The part surrounded in red is the ScriptPubKey of the unspent transaction output.

A scriptPubKey can be seen in the outputs; it represents the conditions that are set for spending the outputs. The new owner can sign using the private key associated with the address receiving the output to fulfil the conditions of scriptPubKey. The network checks whether then if the digital signature is valid, and thus makes it an input for the new transaction. The cryptographic parts - scriptSig and scriptPubKey - are particularly complex and will be discussed in the next chapter.

Copy the `ScriptPubKey` from the output and pass it along the other options as indicated below to the command `signrawtransaction`:

```
signrawtransaction "Raw hexstring" (
[{"txid":"id", "vout":n, "scriptPubKey":"hex", "redeemScript":"hex"},..] ["privatekey",..])
```

The second argument is a JSON array of previous transaction outputs we are consuming, whereas the third argument is the private keys belonging to the address that received the output. The result would be similar to the following output :

[illegible]

After succeeding in signing the raw transaction, it is time to send the signed transaction to the testnet network.

## Sending the transaction

To send the transaction into the blockchain we submit the signed signature using `broadcast` command provided by Electrum as shown in the following picture.

[illegible]

You'll get back the hex-encoded transaction's hash ID:

d3e300c2f2eedf673ab544f4c2b09063353e618ab8a0c9444e931d0145e43ded

# Retrieve your message online from the blockchain

If everything goes as planned, you should have successfully stored the "hello world" message into the Bitcoin's testnet blockchain. The following picture illustrates what we have done so far. We consumed an input (from a previous transaction) then we have created a transaction with two outputs, the first being an `OP_RETURN` transaction carrying our message along, the other one transferring 1 bitcoin (BTC).

⚠ Data Embedded In Transaction with Unknown Protocol (what's this?)

Spring: hello world

Hqx: 00070x0000/007707700004

AMOUNT TRANSACTED

1.0 BTC

FEES

0.1 BTC

RECEIVED

⌚ about 11 hours ago

CONFIRMATIONS ⓘ

🔒 6+

Details

1 Input Consumed

1.1 BTC from

👤 3KkUj0KJ58mm78m94eyedUSky4d3y (local)



2 Outputs Created

0.0 BTC NullData transaction

1.0 BTC to

👤 3MsJ5s4C11Xsah75cs3jmmP4W5hNqpuRq9U (unspent)

Isn't it just fascinating? You can use a block explorer like <https://live.blockcypher.com/btc-testnet/tx/<txid>> to inspect the transaction with the printed transaction hash (txid) and to retrieve your stored message.

It would be more exciting if you retry the same operation using the Mainnet (the original and main network for Bitcoin), but in this case you will deal with real, expensive Bitcoins.

# Retrieve your message from the local blockchain

If the bitcoin core client has finished syncing the blockchain you can locally parse the blocks to locate our transaction and read the stored message .

To open and parse the blockchain blocks we need to install a graphical hex editor like `bless`

```
sudo apt-get install bless
```

Once installed you can run it and open one of the `.blk` files present in the `blocks` directory.

As shown in the following picture, `bless` will display a pane divided into three parts:

- The left column is the offset column.
- The center column displays the blocks hexadecimal content.
- The right column is the same line of data as in the center with recognized text characters displayed as text and binary values are represented by period characters.



the transaction by searching for the previous block delimiter called magic bytes represented by 0b110907. Then you can by following the structure of the block, determine the meaning of these long hexadecimal strings. In the previous picture, I have delimited the block with a yellow border and highlighted the field of the blocks header with different colors and delimited our transaction and the coinbase transaction in blue and gray respectively.

*As you'll be running in the pruned mode you will not be able to see my transaction as you will have only synced newer blocks. However you'll be able to see your transaction by following the same process.*

To help you visualize the block content the following table presents in order the meaning of the previous highlighted bytes:

| Bytes                |                        | Size      |
|----------------------|------------------------|-----------|
| Magic number         |                        | 4 bytes   |
| Block size           |                        | 4 Bytes   |
| Block Header         | Version                | 4 Bytes   |
|                      | Hash of previous block | 32 Bytes  |
|                      | Hash Merkle root       | 32 Bytes  |
|                      | Timestamp              | 4 Bytes   |
|                      | Bits                   | 4 Bytes   |
|                      | Nonce                  | 4 Bytes   |
| Transactions Counter |                        | 1-9 Bytes |
| List of Transactions |                        | -         |

And that's it! You can now send transactions with extra messages into the blockchain, and retrieve the data online or locally. Although this is not usually required, it may prove useful in the future. Let's go ahead and send another RAW transaction with an OP\_RETURN output by writing few lines of code.



## Method 2 : Build a raw bitcoin transactions in JavaScript .

At this level, I would guess that you want to write some code. *Your wish is my command.*

In this section we will build a simple NodeJs script to perform what we have performed manually before: to send a raw transaction over the Testnet network. Henceforth, you can stop Electrum and bitcoin core clients, as we will use an online REST API ([chain.so/api](https://chain.so/api)) as a middle tier to interact with Bitcoin's network.

*By using an online API, we are losing the biggest advantage of blockchain; disintermediation. We have to trust a middleman instead trusting our own blockchain copy. We will ask instead the third party to read the data for us and send the transaction on our behalf. Think of it that the service provider might provide us wrong or outdated data.*

# Preparation

Before you start building your program make sure you have NodeJs and NPM (Node Package Manager) installed.

In order to create an `OP_RETURN` transactions we can use one of many Bitcoin APIs, to name a few:

- `bitcore` (<https://bitcore.io/>),
- `php-OP_RETURN` ([https://github.com/coinspark/php-OP\\_RETURN](https://github.com/coinspark/php-OP_RETURN)),
- `python-OP_RETURN` ([https://github.com/coinspark/python-OP\\_RETURN](https://github.com/coinspark/python-OP_RETURN))

In our example we will use a JavaScript library called `bitcoinjs-lib` written for NodeJs . We install the corresponding package as follows:

```
npm install bitcoinjs-lib --save
```

In the example code, we will submit requests using NodeJs and the Request package to access the API. Therefore, we install the following modules:

```
npm install request --save  
npm install request-promise --save
```

Similarly to the first method, we will use the first address and its corresponding private key generated by Electrum to send programmatically a raw transaction carrying a "hello world" message.



# Let's code

Start by creating a `hello.js` file and importing the `bitcoinjs-lib` and `request-promise` modules using the `require` directive as follows :

```
var bitcoin = require('bitcoinjs-lib');  
var rp = require('request-promise');
```

Then we declare and define the needed variables:

```
var data = Buffer.from('Hello World', 'utf8');  
var testnet = bitcoin.networks.testnet;  
var privateKey =  
'cQx4Ucd3uXEpa3bNnS1JJ84gWn5djChfChtfHskRaDNZQYA1FYnr';  
var SourceAddress = "n3CKupfRCJ6Bnmr78mw9eyeszUSkfyHcPy";
```

They represent respectively:

- The message to embed in the transaction.
- The used network: testnet.
- The private key in WIF format (WIF is an abbreviation of Wallet Import Format).
- The source address from which we spend the UTXO.

Then we request the API to provide us with the available unspent output belonging to a specific address. We read the response from the API to define the available amount and the output txid. Besides this

we define the fee (5000 satoshis) to pay the network (miners) for processing the transaction as follows:

```
var url =  
"https://chain.so/api/v2/get_tx_unspent/BTCTEST/"+SourceAddress;  
var DestinationAddress = '2MsHsi4CHXsaNZSq5krnrpP4WShNgtuRa9U';  
var options = {  
  uri: url,  
  json: true  
};  
  
rp(options).then(function (response) {  
  var index = response.data.txs.length - 1;  
  console.log(response.data.txs[index]);  
  var UtxoId = response.data.txs[index].txid;  
  var vout = response.data.txs[index].output_no;  
  var amount = Number(response.data.txs[index].value*100000000);  
  var fee = 0.0005*100000000;  
}).catch(function (err) { console.error(err);});
```

You can at any level use `console.log()` to print the received values in the console.

Now it's time to create our transaction. Inside the previous GET request add the following lines:

```
const RawTransaction = new bitcoin.TransactionBuilder(testnet);  
RawTransaction.addInput(UtxoId, vout);  
RawTransaction.addOutput(DestinationAddress, parseInt(amount-fee));  
script = bitcoin.script.compile([bitcoin.opcodes.OP_RETURN, data]);  
RawTransaction.addOutput(script, 0);
```

Here we are using `bitcoinjs-lib`'s `TransactionBuilder` to create our new raw transaction, then we add the output we requested earlier from the API as input to our transaction. We add two outputs, the first is an `OP_RETURN` output with 0 bitcoin, and the second is the output with 100000000 satoshis (1BTC) minus the fees.

Great! Everything is set! The only thing we have to do right now is to sign the transaction with our private key and send it to the Bitcoin blockchain :

```
var keyPair = bitcoin.ECPair.fromWIF(privateKeyWIF, testnet);  
tx.sign(0, keyPair);
```

The second line - `tx.sign(0, keyPair)` - is because we are consuming a P2PKH output. However, in bitcoin we have different types of transaction and addresses. The addresses starting with "2" receives Pay-to-Script-Hash (P2SH) transactions instead of the common Pay-to-Public-Key-Hash (P2PKH) transactions received by addresses starting with "m" or "n". This of course changes the way we spend the output; therefore, we need to know the type of the output prior to signing the new transaction. For P2SH transactions we need to use instead the following code :

```
const p2wpkh = bitcoin.payments.p2wpkh({ pubkey: keyPair.publicKey,  
network: bitcoin.networks.testnet });  
const p2sh = bitcoin.payments.p2sh({ redeem: p2wpkh, network:  
bitcoin.networks.testnet});  
RawTransaction.sign(0, keyPair, p2sh.redeem.output, null,  
parseInt(amount));
```

Lastly, we take the signed transaction in and send it to the specified network using a POST request with the API. We provide in our request a JSON object which contains a hex representation of the signed transaction:

```
var Transaction=RawTransaction.build().toHex();  
var Sendingoptions = { method: 'POST', url:  
'https://chain.so/api/v2/send_tx/BTCTEST',  
body: {tx_hex: Transaction}, json: true};  
  
rp(Sendingoptions).then(function (response) {  
    var Jresponse = JSON.stringify(response);  
    console.log("Transaction ID:\n"+Jresponse);  
}).catch(function (err) { console.error(err); });
```

Once you have saved the file, run it with the command `node hello.js`. If the raw transaction is valid and delivered successfully to the network, you will receive back a message similar to the following:

```
user@ByExample-node:~/deletme$ node hello.js
{ txid: '0615fe519f4583546ea9d39d3ca7628e0ea56cd3037ea64d1bb0e967f48d93e4',
  output_no: 0,
  script_asm: 'OP_DUP OP_HASH160 edcce89f510bf95606ec6a79cb28a745c039e220 OP_EQUALVERIFY OP_CHECKSIG',
  script_hex: '76a914edcce89f510bf95606ec6a79cb28a745c039e22088ac',
  value: '1.10000000',
  confirmations: 0,
  time: 1533938378 }
Transaction ID:
{"status": "success", "data": {"network": "BTCTEST", "txid": "f917abfa314ebd774a4a659c69c924cab3d8e4e9f4f41ccd7620a985b373a2fb"}}
```

We get the used output details along with the success message returning the transaction ID. As we did before, we can check the transaction processing using a Testnet explorer.

Congrats, you have successfully built your first NodeJs application to send bitcoins and to store data into bitcoin's blockchain. Based on that you can create advanced applications or develop your own protocol on top of the Blockchain. As a bonus the full code is available on the following Github repository: <https://github.com/bellaj/HelloWorld>

# Types of blockchains

The blockchain had started initially as the technology underpinning bitcoin. However, its early success to prove itself as trust machine and intermediaries' killer, has put pressure on a variety of industries to both adapt and adopt this new technology. Consequently, many variations of the original Satoshi's blockchain have been developed. In this final section we will take a look at the major types blockchain technologies.



# Classification of blockchains

Currently, blockchain solutions could be classified into four basic models with different paradigms. The distinction between them is driven by two main criteria : the network type and the implemented access control model. In other terms who is allowed to join the peer-to-peer network (run a node) and access the blockchain records. The following table gives an overview of the different types with a basic description of each models :

| Model                           | characteristics  | Technologies  | Strengths  |
|---------------------------------|--|---|--|
| Public blockchain               | The general public can join the network and write (under consensus protocol) and read data. This model is a true representation of the original blockchain used in the cryptocurrencies.   | Bitcoin, Zcash, Ethereum, Litecoin, NXT, etc.   | Full-Decentralization<br>High Security<br>Censorship resistant<br>Low-trust<br>Anonymity<br>Transparency |
| Private or internal blockchain  | A network under the governance of one organization, which defines the access rules to join its private network. Therefore, only authorized entities can read the transactions data. The organization defines nodes with high trust levels to accept the transaction.   | Monax, Symbiont Assembly, Iroha, Kadena, Chain, Quorum, MultiChain.                                   | Confidentiality<br>Authenticated parties<br>Privacy<br>Faster<br>Less expensive                          |
| Consortium or hybrid Blockchain | A partly private and permissioned blockchain network operated by known entities such as stakeholders of a given industry regrouped in a consortium or exploiting a shared platform. The network participants have control over who can join the network, and who can participate in the consensus process of the blockchain. | Hyperledger Fabric, Tendermint, Symbiont Assembly, R3 Corda, Iroha, Kadena, Chain, Quorum, MultiChain | Confidentiality<br>Authenticated parties<br>Privacy<br>Faster<br>Less expensive                          |
| Blockchain as a service (IaaS)  | Cloud platform hosted by a service provider to deploy blockchain applications. The service provider manages the blockchain network while the customer defines the business logic.  | Bluemix, Azure, Kubix, Stratis, AWS, SAP, Oracle.   | Flexibility<br>Scalability<br>Complexity reduction   |

You might be wondering which one amid these types of blockchain would be appropriate for your project?

The answers depend on your project's requirements. Generally, in a trustless environment we tend to use a public blockchain (remittance

systems, provenance, etc.), whilst the other models are fit for actors who share a pre-existing trust and desire to build shared services. Private blockchains are suitable for building faster local testing environments or to avoid the cost of transacting, whereas the BaaS model is a suitable solution for easy deployment and high scalability.

# Summary

This first chapter is meant to acquaint you with key concepts behind the blockchain that you'll need to build blockchain projects. We have seen how you can manually and programmatically (using JavaScript) interact with bitcoin's blockchain. This first achievement can help you start building innovative solutions harnessing the blockchain as a safe and trustworthy vault, such as: checking the authenticity of documents, proving the provenance of products, asset digitization, etc.

Understanding Bitcoin deeply will be the most important weapon in your arsenal to decipher the Blockchain. Given that, in the next chapter, we will continue learning about bitcoin and raise the bar higher to build an advanced payment application.

## **Chapter 2 : Building a Bitcoin payment system**

# Introduction

The previous chapter was an excellent starting point for understanding the blockchain and learning about Bitcoin. In fact, we covered many basic concepts and elementary operations such as signing, sending raw transactions and storing data into the blockchain. We'll herein, continue practically exploring the blockchain, by first building a customer-friendly payment system based on the Bitcoin payment protocol and secondly build our first smart contract in the Bitcoin system.

The end goal of this chapter is to help you acquire the necessary technical background for understanding Bitcoin's mechanics, and build your first Bitcoin applications using very common languages such as JavaScript and Java. If you're an experienced programmer, most likely you are familiar with one of these technologies, otherwise I would recommend you spending time reading their official getting started documentation.

Throughout this chapter we will cover the following key points:

- Introducing bitcoin.
- Building BIP 70 payment system using `Nodejs` and `bitcore-lib`.
- Building bitcoin client using `BitcoinJ`.
- Writing and deploying Smart contract in bitcoin using `Rootstock`.

This is a fully hands-on coding chapter, I will insist that you code each of the snippets presented herein to make sure you get the most

out of the chapter. Happy coding!.

# What is Bitcoin?

As outlined in the previous chapter, Bitcoin is a peer-to-peer electronic cash system based on blockchain technology . Technically speaking, Bitcoin is a protocol maintaining the blockchain data structure and ensuring consensus between different network parties (sender, recipient, miners, etc.). This protocol defines the ruleset for validating the transactions, bitcoin minting, and to avoid counterfeiting or double spending.

Bitcoin is referenced as a cryptocurrency because it uses cryptography to control the creation and transfer of money. Specifically, it uses digital signatures (based on ECDSA public key encryption) to process and verify the transactions and cryptographic hash functions (SHA-256) to secure the blockchain (integrity, mining process, etc.).

This book's scope isn't about presenting the Bitcoin protocol in fine detail but instead we will introduce the needed concepts for building higher level services. If you are eager to learn deeply about bitcoin, you can refer to the official [documentation](#).



# Why choose Bitcoin as platform?

Although using bitcoin directly to build blockchain applications might seem to you somehow obscure, it still can be the best choice to build powerful applications. Apart from the fact that blockchain and Bitcoin are hot trends, several factors can motivate you to choose bitcoin as a platform for your next project, including :

- Bitcoin is an electronic payment pioneer and the most accessible system for users worldwide.
- Settlement processing is faster than many payment channels.
- Transaction are without chargebacks.
- Bitcoin is the most secure blockchain and more secure than the common payment channels.

I am proposing in the following section you begin with a simple and easy-to-follow practical introduction before we start developing our first application.

# Getting started with Bitcoin

To get started using Bitcoin there exist plenty of Bitcoin clients and different implementations of the Bitcoin protocol. As we have seen in the previous chapter, the very common used Bitcoin client is Bitcoin Core which is maintained by the Bitcoin Core team representing the C++ implementation of the Bitcoin protocol and the continuity of the Satoshi client. I'll assume that you have already installed this client and have become familiar with it.

# Setting up regtest environment

For learning or testing purposes, it's safer and cheaper to use Bitcoin's test network `--testnet` or regression test mode `--regtest`. We learned in the previous chapter how to use the testnet network but this option still needs us to download the blockchain and wait for delayed validation. However, there is a solution, and this time we are going to use the Bitcoin client in `regtest` mode.

In this mode, we can setup a local testing network and a private blockchain whereby we can instantly validate transactions and locally create new bitcoins. Similar to running a web application in localhost mode, you'll find the `regtest` option more suitable to develop and test new applications.

You have to follow the same procedure as in the previous chapter and add an extra step: change the `bitcoin.conf`, to select the `regtest` network by defining the parameter `regtest=1`.

Your `bitcoin.conf` will look like:

```
rpcuser=<user>
rpcpassword=<password>
#testnet=1
#prune=550
regtest=1
server=1
```

We commented out `testnet` and `prune` parameters as they are no longer needed for the `regtest` mode. In addition, make sure to replace RPC

credentials <user/password> and use them throughout this section when they are needed.

*If you want to build a network with multiple bitcoin instances instead of a single node on your machine, you have to define different data directories with different `bitcoin.conf` files. In each you'll need to set the new path and different communication ports using the parameters :*

*-datadir  
-port (default: 8443 )  
-rpcport(default: 18442)*

You can find more information about the bitcoin regtest mode in the [official documentation](#).

# First commands

For the sake of simplicity we will setup a private blockchain with a single node. Start by opening two separates command line prompts. In the first run `bitcoind` (bitcoin node server) while in the other run one of the available RPC command using `bitcoin-cli <command>`. To get the full list of RPC calls you can use the help option `bitcoin-cli --help` or by visiting the official documentation: [https://en.bitcoin.it/wiki/Original\\_Bitcoin\\_client/API\\_calls\\_list](https://en.bitcoin.it/wiki/Original_Bitcoin_client/API_calls_list).

As a first command we will generate quickly 101 blocks by running:

```
bitcoin-cli generate 101
```

The expected output is a list of created blocks' IDs, similar to the following:

```
user@ByExample-node:~/deletme$ bitcoin-cli --regtest generate 101
[
  "193a6c217cf7918c1f8b040c415691c131721000746d206c37fc347ede25b1f0",
  "44b3f041bef18fc1265268a752834f5456962b0e7ffe7b2584dd87cd194e0fcc",
  "70edb90d0b3cd510998d15d882b340486dd30eac04bc3650c90d31883f771736",
  "76357d14d35adf2f79b562a6cfa2902defe73fb87efdd115deef2179143aa858",
  "1f1e00f9526fda59632943c95838caed9ee338eef86aa2db2a1ed8dc6e7a794a",
  "347250ca8ed505267dde9f305950dc4e8fd69de37c132f8066a66fa6b0d81168",
  "64f4ca37116dbfbabbdd832a81f469bfc7d83e0856952861344ff05ab2ff1df1",
  "69b3db538b64303c42a8ccc7d2ac1d433f03c99803d9623eddc6ebdd8958d65c",
  "07fd1a3186d6bbe7d2613b9ca2ff4ba34dc75205c7992e018d947c10d19aa155",
  "09f31ebf056c4d38070b8dfcd243440de58e299d797464c97e8bb488b49d543b",
  "2d3ed1cad6b92de823f2a7d7f9ed2771f6859a54cf610f856a9bb7fb7e6d0e6c",
  "744db4f8983a5f16c88f723754186289c55a5e985f9f4ac4c5dd71cfe27f7d24",
  "1ac745d9a88a4ef11fd8c64c5ba3a263bcc22694923c32d1483834d60e8df319",
  "31c9709fd9a601688f9ed652dc9034512eb19b956052afd564b16e7b8e57b699",
  "0627f6a23c017f073cb58a0e87acc218aec2fc5ef84cde0f69fadccbe454630",

```

It is worth noting that in regtest mode, you have to generate 100 blocks (100 confirmations) to get the initial reward (50 bitcoins) generated for the first block. We can check out how many bitcoins we have with:

```
bitcoin-cli --regtest getbalance  
50.00000000
```

Instead of interacting directly using bitcoin-cli you can run commands via the HTTP JSON-RPC tools. If CURL isn't installed, you can install it with `sudo apt-get install curl`. For example you can request your balance by using CURL :

```
curl --user user:password --data-binary '{"jsonrpc": "1.0",  
"id": "curltest", "method": "getbalance", "params": [] }' -H 'content-  
type: text/plain;' http://127.0.0.1:18443
```

Notice the presence of the RPC username and password defined earlier in bitcoin.conf, used with the option `--user` in the RPC call.

*You can also use REST calls to communicate with your bitcoin node. However, in this case you'll need to enable the REST API by adding the option `rest=1` to your bitcoin.conf file. The available calls are presented in the official documentation: <https://github.com/bitcoin/bitcoin/blob/master/doc/REST-interface.md>*

After successfully setting up the regtest environment, what follows is a small introduction to transactions in bitcoin with some basic knowledge required to build the project.

# Transactions in Bitcoin

Without delving into the working internals of bitcoin in deep detail, we need to learn more about the following key concepts :

- Mining.
- Scripts.

Let's take a look at each one in detail in the following sections.

# Mining

When a transaction is sent to the bitcoin network it is not finalized until it gets included in a "block" of transactions by a "Bitcoin miner". Being a miner isn't a reserved role but an open position for anyone able to provide enough computing power to validate the transactions. All the time, the miners in the network are racing to be the first one to validate a block of transactions by performing a difficult computational operation to solve a function defined by the protocol (Proof of Work). The first miner to succeed is rewarded with a prize of newly generated Bitcoins along with "transaction fees" payed for each transaction, as well as his blocks being included into the blockchain.

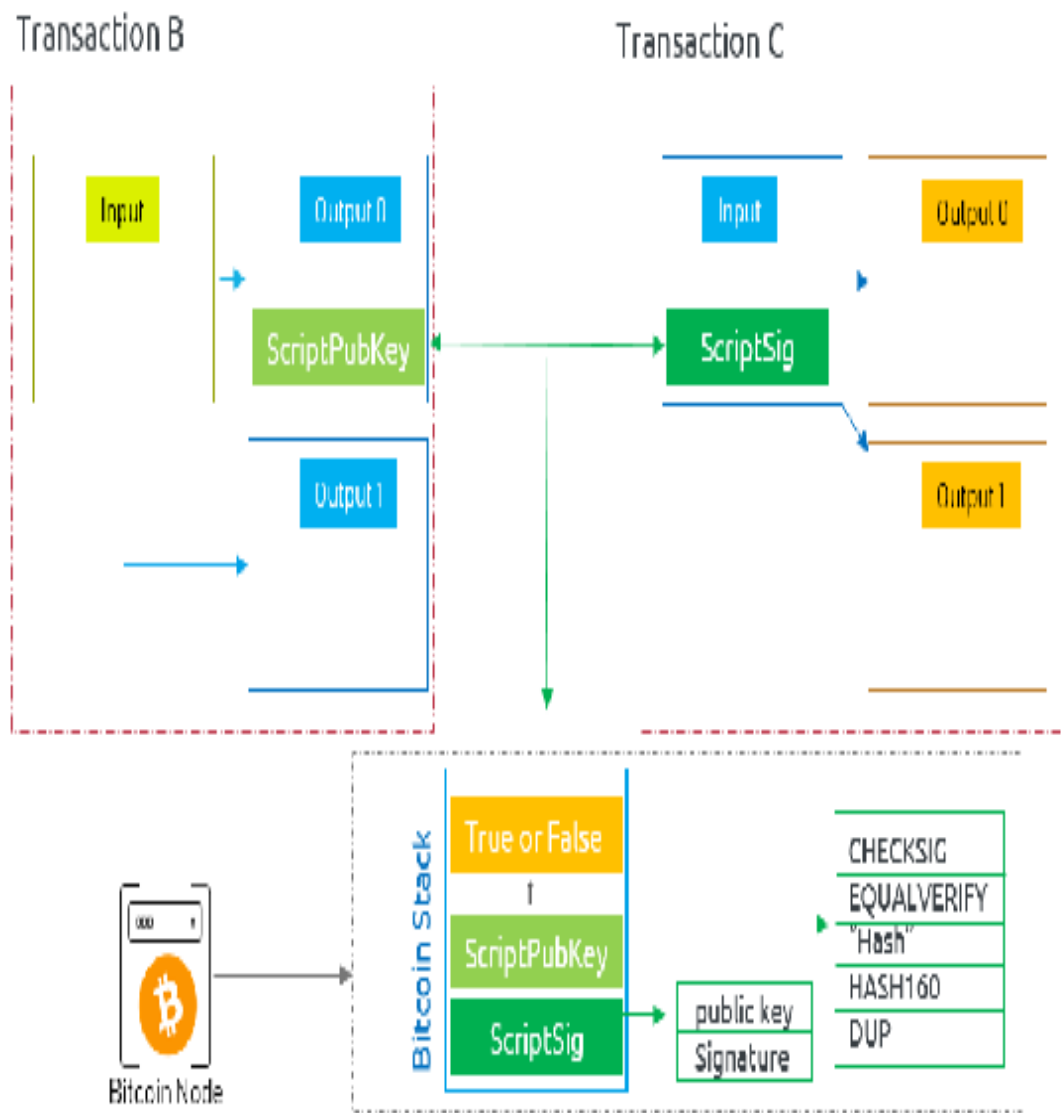
However, it should be noted that these mechanisms are susceptible to be reversed hence the need for waiting for a few confirmations (more than 6 blocks) to consider the transaction as final.



# Bitcoin Scripting

One of the amazing features in the Bitcoin system is the ability to set a script defining the conditions that a recipient should validate to spend the bitcoins later, making bitcoin a programmable currency. Fundamentally, all bitcoin transactions have scripts, written in the Bitcoin programming language, included in their inputs and outputs. This language is a Forth-like language offering a set of opcodes or instructions, evaluated from left to right using a stack to determine the success or the failure of the script execution.

Normally, the transaction embeds into its inputs an unlocking script commonly called `scriptSig` and into the outputs a locking script called `scriptPubkey`. When a transaction is validated, the concatenation of both scripts – `Scriptpubkey`, which protects the output, and the `Scriptsig` provided by the recipient to prove ownership – must execute successfully (evaluated to true). The following figure illustrates the location of both scripts and how they are validated:



Bitcoin scripting using different combinations of opcodes enables us to create a wide variety of transaction types. The following table summarizes the standard transactions types :

| Script Type                    | Script Format |  | Description  |
|--------------------------------|---------------|--|--|
| Pay to PubKey Hash (P2PKH)     | ScriptPubKey  | OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG                      | This script is the most popular type in the Bitcoin blockchain. It is used to send Bitcoins to a single address (which is a public key hash encoded in base58check starting with 1).                                     |
|                                | ScriptSig     | <signature> <PubKey>   |  |
| Pay to PubKey (P2PK)           | ScriptPubKey  | <Public Key> OP_CHECKSIG   | This script is usually used by miners to redeem their new mined Bitcoins.  |
|                                | ScriptSig     | <signature>  |  |
| Pay to Script Hash (P2SH)      | ScriptPubKey  | OP_HASH160 <script hash> OP_EQUALVERIFY  | A special script which moves the responsibility for supplying the conditions to redeem a transaction from the creator of the transaction to the receiver(s). This latter must provide a script matching the script hash. |
|                                | ScriptSig     | <signature> <serialized script>  |  |
| Multi-Signature                | ScriptPubKey  | ScriptPubKey=2 <Public Key A> <Public Key B> <Public Key C> n OP_CHECKMULTISIG | This script locks the output with N public keys, with condition that at least M of those Keys are required. The redeemers should provide M valid signatures corresponding to the N public keys.                          |
|                                | ScriptSig     | <Signature1><Signature2> or any permutation of Sig m:n                         |  |
| P2WSH                          | ScriptPubKey  | zero [32 byte sha256(witness script)]  | P2WSH is similar to P2SH but the proof of ownership RedeemScript is moved to the witness instead of ScriptSig (signature + redeem script).   |
|                                | ScriptSig     | Empty (as the signature is located in the witness instead of the ScriptSig)    |  |
| Pay to Witness Public-Key-Hash | ScriptPubKey  | ScriptPubKey: zero [20 byte hash160(public key)]                               | P2WPKH is similar to a P2KH but with witness data.   |
| Null Data:                     | ScriptSig     | Empty  | This script marks a transaction as provably unspendable and a potential prunable Output. In this case the script is considered as invalid, guaranteeing that there is no corresponding ScriptSig.                        |
|                                |               | OP_RETURN OP_DATA_X  |  |

Lastly, it's worth mentioning that users can define their own locking Scripts but they should request miners to mine them.

*If you're familiar with C++ you can understand how scripts works under the hood by looking at script interpreter code: <https://github.com/bitcoin/bitcoin/blob/master/src/script/interpreter.cpp>. Moreover, a transaction is considered as standard if it fulfils the requirement defined by Bitcoin Core's `IsStandard()` and `IsStandardTx()` functions.*

We have now clarified the role of the scripts but they will make more sense the more practice you get. Let's take a look at an example of how to build and send bitcoin transactions with custom scripts using JavaScript.

# Building p2pkh script using JavaScript

The first hands-on part of this chapter will be writing a JavaScript snippet using the powerful Bitcoin library `bitcore-lib` to build a P2PKH script (used to pay to a bitcoin address). Throughout this chapter, we will run our experiments on Ubuntu LTS 16.04.

Before you proceed with this section, make sure NodeJs ([nodejs.org](https://nodejs.org)) is installed with the latest version. Then create a directory in which we install the bitcore package :

```
npm install bitcore-lib --save
```

Let's take a look at how to build very simple script:

```
var bitcore = require('bitcore-lib');
var Address = bitcore.Address;
var address = Address.fromString('n3CKupfRCJ6Bnmr78mw9eyesZUSkfyHcPy');
var script = bitcore.Script.buildPublicKeyHashOut(address);
console.log(script);
```

The `bitcore.Script` object provides an interface to construct Bitcoin scripts. It also gives simple interfaces to create the most common script types such as `buildPublicKeyHashOut(address)` which creates a Pay-to-Public-Key-Hash output for the given address.

Save the code in a JavaScript file `script.js` and run it using `node script.js`. The output will show you the following `ScriptPubKey` script (locking script) :

```
<Script: OP_DUP OP_HASH160 20  
0xedccea89f510bf95606ec6a79cb28a745c039e220 OP_EQUALVERIFY OP_CHECKSIG>
```

The result is a common locking script requiring the receiver to have the private key corresponding to the public key whose Hash160 ( RIPEMD160(SHA256(publickey))) is 0xedccea89f510bf95606ec6a79cb28a745c039e220. This special hash was extracted from the receiver's bitcoin address as:

Bitcoin address = [Version Byte (1 byte)]+[Hash 160 of public key]+[Checksum (4 Bytes)]

This is then encoded in base 58. If the receiver provides the public key, he has to prove also his ownership of the right private key by signing the transaction to fulfill the OP\_EQUALVERIFY OP\_CHECKSIG part.

# Building a custom script

As mentioned before, we can define our own scripts instead of using the standards. In the following example, we will define a non-standard Bitcoin Script based on a simple equation  $X+13=15$ . Therefore, to spend our transaction the recipient needs to come up with a Scriptsig presenting the right solution which is obviously "2" to solve the equation and spend the output.

If we translate this equation into bitcoin scripts we get :

|   |                                  |
|---|----------------------------------|
| Locking Script (ScriptPubKey):<br>"x+13=15" | OP_X OP_13 OP_ADD OP_15 OP_EQUAL |
| Unlocking Script (ScriptSig): "2"           | OP_2                             |

Bitcore enables us creating transactions with custom script. Hence, in the following example we will create a non-standard transaction with the puzzle (Scriptpubkey) described before.

Firstly, we need to select a UTXO from our Bitcoin wallet to construct a new transaction. For that, make sure bitcoin core is still running in regtest mode and use the command `bitcoin-cli listunspent` to get an available UTXO with its details : `txid`, `Scriptpubkey` and receiving address. Then create a new destination address with :

```
bitcoin-cli getnewaddress.
```

On the other hand, the private key can be unveiled using `bitcoin-cli dumpprivkey <utxo address>`. Once you have all these ingredients, edit the following code accordingly (full code is available at [https://github.com/bellaj/Bitcoin\\_payment/tree/master/custom%20scripts](https://github.com/bellaj/Bitcoin_payment/tree/master/custom%20scripts)):

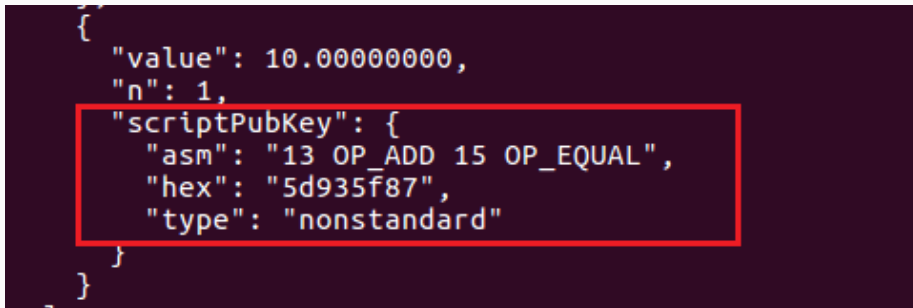
```
var pkey = 'cPJct9r5eu9GJz1MxGBGmZYTymZqpVVCZ6bBdqYQQ5Pew4h74d'; //UTXO's private key
var Address = 'n1PoDECeUwbXgktfkNkBcmVXtD2CYUco2c'; //Destination address
var lockingscript = bitcore.Script('OP_13 OP_ADD OP_15 OP_EQUAL'); //PubKeyScript
var g_utxos=[{"address":"n1PoDECeUwbXgktfkNkBcmVXtD2CYUco2c",
"txid":"c6758cf22346d3d8b7b6042b7701a5f07d140732bf5b93e1fb92ed250e5b6d20","vout":0,"scriptPubKey":"210330b8e88054629399e6c9
//UTXO details

var transaction = new bitcore.Transaction();
transaction = transaction.from(g_utxos);
transaction = transaction.to(Address, 4000000000); //Add a first output with the given
amount of satoshis
transaction = transaction.fee(0.0001*100000000);
transaction = transaction.addOutput(new bitcore.Transaction.Output({script:
lockingscript, satoshis: 1000000000,address:Address }));
transaction = transaction.sign(pkey); //Sign all inputs
console.log("Raw Transaction\n" + transaction);
```

---

The previous code is self-explanatory: we construct a transaction using the `Transaction()` method, then we define two outputs – one sending 40 BTC to a P2PKH address and the other sending 10 with a custom script.

Save this code in a file called `custom_pubkeyscript.js` and run it using `node custom_pubkeyscript.js`. As a result, a raw transaction will be constructed. If you decode the resulting transaction using `bitcoin-cli decoderawtransaction <your tx>` you'll be able to see our custom ScriptPubkey :



```
{
  "value": 10.00000000,
  "n": 1,
  "scriptPubKey": {
    "asm": "13 OP_ADD 15 OP_EQUAL",
    "hex": "5d935f87",
    "type": "nonstandard"
  }
}
```

Then send the raw transaction to the local network using `bitcoin-cli sendrawtransaction <your raw transaction>` you'll get back the transaction ID. To get it validated we have only to mine a block using `bitcoin-cli generate 1`

Now we have a non standard output carrying 10 bitcoins waiting to be spent. To consume it, we need to construct a transaction with the correct ScriptSig `op_2` as follows:

```
var unlockingScript = bitcore.Script().add('OP_2');
var transaction = new bitcore.Transaction();
transaction.addInput(new
bitcore.Transaction.Input({prevTxId:'c6758cf22346d3d8b7b6042b7701a5f07d140732bf5b93e1fb92ed250e5b6d20',
  outputIndex: 1, script: unlockingScript }), unlockingScript, 10000);
transaction = transaction.to(Address, 90000000);
transaction = transaction.fee(0.0001*100000000);
console.log(transaction)
```

We define here an input pointing the previously created output with a custom scriptSig. As you may notice, the output can be spent without providing a signature. As we did for the previous transaction you can send the transaction and mine it.

When the ScriptSig and Scriptpubkey are executed, the opcode `.add('OP_2')` pushes value 2 onto the stack, then the operands (13 and the 2 from ScriptSig) are added using `.add('OP_ADD')` and the result is compared using `.add('OP_EQUAL')` to 15 therefore the top on the stack will be true which means the full script (unlocking + locking) is valid and the output is "unlocked" and can be spent.

*To observe the execution of bitcoin script on the stack., there is an interesting open source IDE for Bitcoin transaction called [hashma1](https://github.com/mazaclub/hashma1) available on <https://github.com/mazaclub/hashma1>*

If everything went correctly, you should be able to successfully spend the previous output with the custom ScriptPubkey. Just a minor warning, in public networks non standard transactions may not be validated by the network.

All set, we're done with non standard transactions and bitcoin scripting. Now on, you have a basic understanding of bitcoin under your belt, you should be ready to tackle

the rest of this chapter and build real world application.



# Building a payment gateway

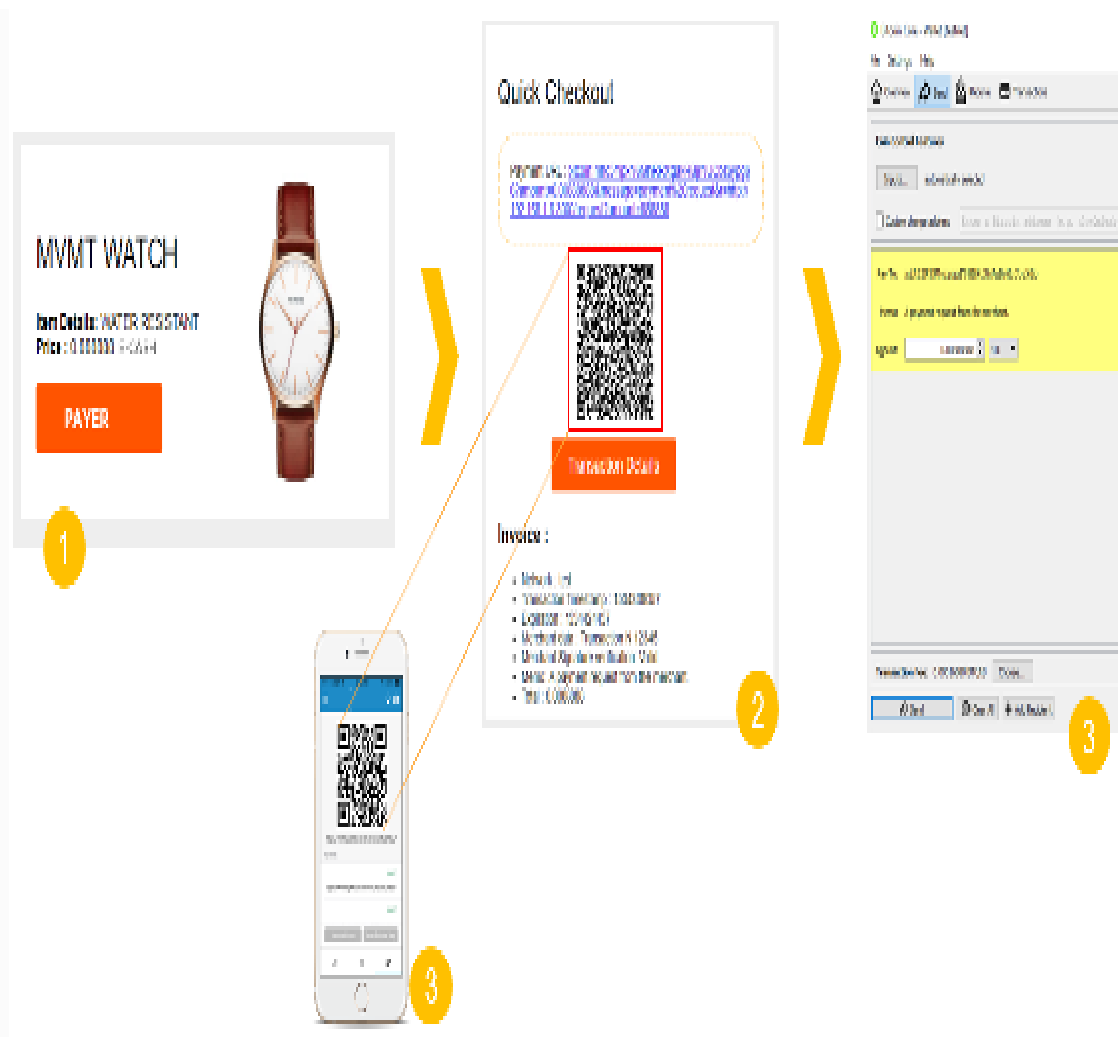
If you have never developed Bitcoin applications before, this section is be the best place to start. We plan here to integrate Bitcoin payment in an online Ecommerce website. The idea is to create a payment option for the E-commerce customers to pay simply by clicking a direct Bitcoin payment URL or scanning a QR code, which opens a payment form with payment details in the wallet, making the payment process very straightforward and easy. Moreover, such a payment option avoids customers filling in any forms or providing personal data.

# Project description

This project generally represents how to implement the Bitcoin payment protocol proposed in [BIP70](#) in order to build an online payment gateway. BIP 70 protocol enables direct Bitcoin payments processing for e-commerce platforms by managing payment transactions between a payment portal and the customer's Bitcoin wallet.

*BIP or Bitcoin Improvement Proposal, is a proposal for introducing features or changes to Bitcoin. You can learn about all the BIPs in the official documentation : <https://github.com/bitcoin/bips>*

The merchant's server application will generate a custom [BIP 72](#) payment URL (and the corresponding QR code) to help customers checkout easily using their Bitcoin wallet/client, whether by clicking on the provided link or by scanning the QR code, as shown in the following picture.



Basically, we'll follow three major steps to setup the project:

1. Building the Merchant server using NodeJs and the bitcoine library.

## 2. Building the Javascript front-end .

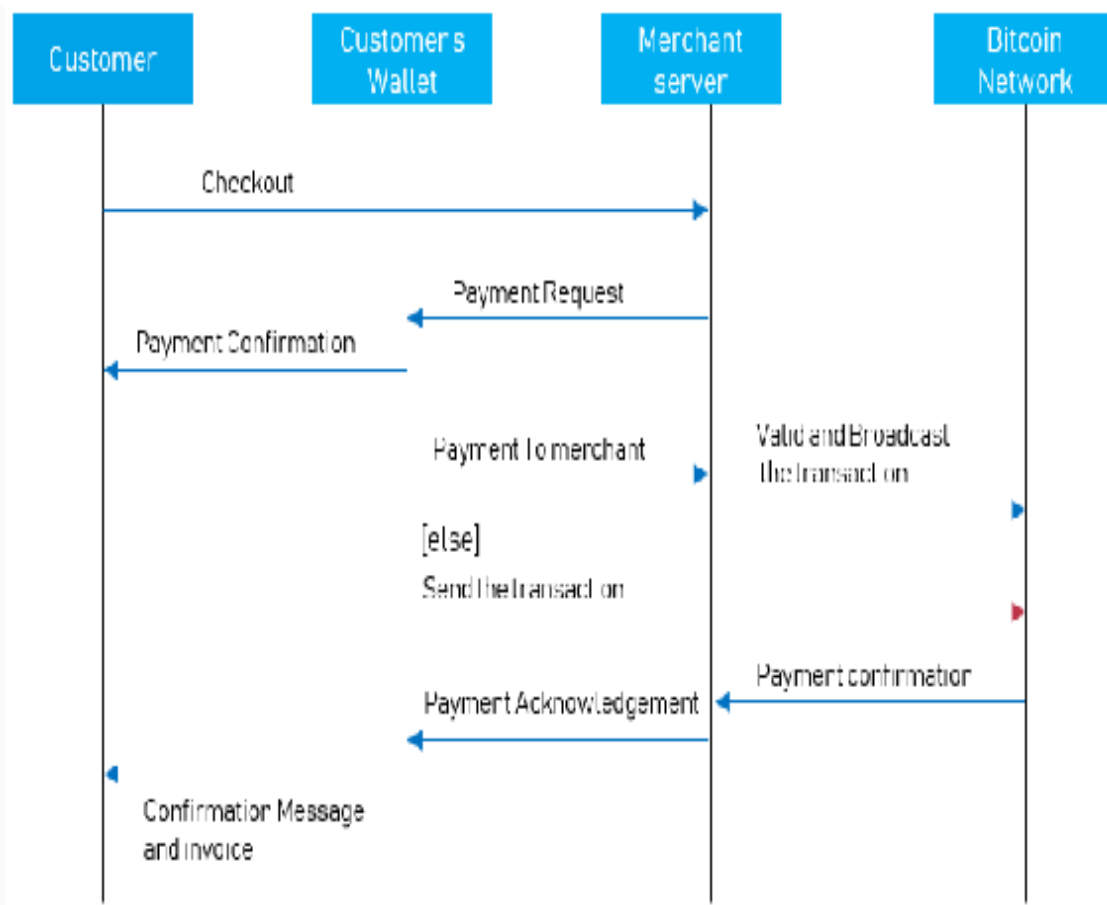
### 3. Building the Java client using BitcoinJ.

# Bitcoin payment protocol overview

The Bitcoin payment protocol specification is presented in BIP 70 (<https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki>), [71](#), [72](#) and [73](#) documents. It was designed to introduce additional features to Bitcoin by replacing the usage of Bitcoin address with direct payment flow between the sender and recipient of funds using graphical wallet to offer seamless checkout process. The main goal of BIP 70 is to provide a Bitcoin payment solution improving the customer experience and securing the online payments.

*If you are familiar with C++ you can examine at the `paymentserver.cpp` and `paymentrequestplus.cpp` located in the bitcoin Github repository <https://github.com/bitcoin/bitcoin/blob/master/src/qt/>.*

The user does not have to deal with Bitcoin's underlying mechanisms, he has only to simply open his wallet with a Payment Request and proceed with the payment. The sequence diagram below shows how the customer and merchant's wallets interact with each other and with the Bitcoin network to process the payment operation using BIP 70 protocol.



When the customer initiates the checkout process, the merchant launches a payment request to the user's wallet signed with its certificate. Once received, the latter parses and validates the request details then authorizes the payment by sending back the payment transaction to the merchant or directly to the network. When the payment is settled the merchant sends a payment acknowledgement to the user with potential invoice details.



# Prerequisites

Before proceeding with this project, a general knowledge of programming concepts and JavaScript is highly recommended. To start coding, we need to have the following elements installed:

- npm (The npm package gets installed along with NodeJs.)
- bower
- git

For this project, I suggest we switch bitcoin client to operate on the testnet. In your `bitcoin.conf` file keep the same user, password and edit the following values

```
regtest=0  
testnet=1  
prune=550
```

Afterwards, run bitcoin-qt and generate two addresses, one for the merchant and the other for the customer, and then provision some testnet Bitcoins to the customer's address from an online Bitcoin faucet such as <https://testnet.manu.backend.hamburg/faucet>

# Project setup

After fulfilling the prerequisite steps, create a new directory dedicated to your project, as follows:

```
mkdir Bitcoin_payment && cd Bitcoin_payment
```

In your workspace create two new subdirectories : 'keys/' for storing your keys and certificate, and a 'views/' directory to store the web pages and scripts.

To install all the needed dependencies, create a package.json file from inside the 'Bitcoin\_payment/' directory

```
{
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "bitcore-lib": "^0.15.0",
    "bitcore-payment-protocol": "1.2.2",
    "body-parser": "^1.18.3",
    "express": "^4.16.3",
    "request": "^2.88.0",
    "underscore": "^1.9.1"
  }
}
```

Then run :

```
npm install
```

This will install all the Node packages you need for setting up and running the project.

# Certificate management

In order to build a secure payment system we need to use SSL certificates either for enabling HTTPS or for BIP 70 payment request verification to confirm that the payment request was genuinely initiated by the merchant. Basically, the certificate has to be provided by a third-party certificate authority (CA) to confirm the merchant identity.

To obtain SSL certificates, you have the choice between using Commercial or Private Certificate Authorities or the "[Let's Encrypt](#)" project. For demonstration purposes, in this guide we will use a self-signed certificate to avoid unnecessary details. To obtain such a certificate, you need to use OpenSSL as follow:

```
openssl req -x509 -newkey rsa:4096 -keyout keys/key.pem -out  
keys/cert.pem -days 365 -subj  
"/C=MA/ST=ByExample/L=Testbip70.com/O=Chapter3/OU=Org/CN=Testbip70.com"  
-nodes  
  
openssl x509 -in keys/cert.pem -out keys/cert.der -outform DER
```

You will end up with three files: `key.pem` , `cert.pem` and `cert.der` in your `'keys/'` directory.

# Merchant side

To start off, we build a basic NodeJs server accepting HTTP and HTTPS requests using our generated certificates. As this book is not about neither NodeJs nor JavaScript, I'll put the main emphasis on presenting the code related to Bitcoin and the bitcore library. For now, let's start building our web server.

```
'use strict';
var bitcore_lib = require('bitcore-lib');
var PaymentProtocol = require('bitcore-payment-protocol');
var express = require('express');
var bodyParser = require('body-parser');
var URI = require('bitcore-lib/lib/uri');
var request = require("request");
const https = require('https');
var fs = require("fs");

var dcert = fs.readFileSync('./keys/cert.der');
var mcert = fs.readFileSync('./keys/cert.pem'); // For HTTPS server
var mkey = fs.readFileSync('./keys/key.pem');

var credentials = {key: mkey, cert: mcert};
var app = express();
var os = require('os');
var interfaces = os.networkInterfaces();
var addresses = [];
for (var k in interfaces) {
  for (var k2 in interfaces[k]) {
    var address = interfaces[k][k2];
    if (address.family === 'IPv4' && !address.internal) {
      addresses.push(address.address);
    }
  }
}

var IP = addresses[0];
var port = 8883;
var http_port = 3000;
```

```
app.get("/", function(req, res) {
  res.send('Bitcoin Payment protocol');
});

app.listen(http_port, function() {
  console.log("-http Server listening on :"+IP+": "+ http_port);
});

https.createServer(credentials, app).listen(port, function() {
  console.log("-https Server listening on :"+IP+": "+ port);
});
```

This big chunk of code creates a simple nodejs server to serve the HTTP requests. Save this code in a file named `server.js` and run it with `node server.js`. Your server will start and listen on two different ports: 3000 for HTTP and 8883 for HTTPS.

*You might get the error : "Error: More than one instance of bitcore-lib found" in this case refer to [https://github.com/bellaj/Bitcoin\\_payment](https://github.com/bellaj/Bitcoin_payment) , I am proposing there a sketchy workaround.*

Since the certificate is self-signed, if you open any browser to the server's address with port 8883, it will yield a warning, and not allow the communication to continue without an explicit exception. To solve that you have to manually install the certificate (`cert.der`) into your browser. Next, try to visit locally `http://localhost:3000`, the browser window should display "Bitcoin Payment protocol" message.

Before we move on, stop the running server by pressing `ctrl+c` in the server's terminal window. Let's expand our code to make use of the bitcore library to create the payment request.

# Building payment request URI

First off, we define the Testnet as the default network for our application and a merchant address. Simply put within the server's code the following lines :

```
bitcore_lib.Networks.defaultNetwork = bitcore_lib.Networks.testnet; //
the project runs only on testnet
var Merchant_address = "mhc5YipxN6GhRRXtgakRBjrNUCbz6ypg66";
```

In this example we used a static Bitcoin address but in a real implementation the merchant has to generate a unique payment address associated with the customer's order. Alternatively you can generate a random address using:

```
var privateKey = bitcore_lib.PrivateKey(merchant_pkey); // we pass a
specific private key as argument
var publicKey = bitcore_lib.PublicKey(privateKey);
bitcore_lib.Address(publicKey, bitcore_lib.Networks.defaultNetwork );
```

We define afterwards a simple function `compose_uri()` to build the Bitcoin payment URI:

```
function compose_uri(amount_to_pay) {
  var pay_url = "http://" + IP + ":" + http_port + "/request";
  var uriString = new URI({
    address: Merchant_address,
    amount : amount_to_pay, // amount in satoshis
    message: 'payment request'
  });
  var paymentUri = uriString + "&r=" + pay_url;
```

```
    return paymentUri;  
}
```

---

The `compose_uri()` function generates the request payment URI starting with the prefix `bitcoin:` (defined in BIP-21) and containing the destination and amount. The overall URI is a BIP-72-style with the special query parameter `r` which specifies where the payment request will be fetched from. Custom URIs are very helpful as browsers and mobile apps use them to launch the registered protocol handler, in this case a Bitcoin client.

So now that we have all that up and running, it's time to actually handle requests.



# Routing

The actual application functionality is just a basic User model with a few views and a controller without supporting registration and logging in/out. Unsurprisingly, to handle the HTTP requests we are using the `expressJs` framework which provides us with a middleware to handle incoming HTTP requests and translate each into an action.

# Checkout view

The initial endpoint will be `/checkout`, whereby the customer requests to make a payment. We define the handler associated with a `get` handler to `/checkout` as follows:

```
var path = require("path");
app.use(express.static(path.join(__dirname + '/views')));
app.get('/checkout', function(req, res) {
    res.sendFile(path.join(__dirname+'views/index.html'));
});
```

If you're unfamiliar with Express, we defined here a callback function that behaves like middleware to handle the `/checkout` route. We are using `sendFile` to make the server send a static `index.html` file from the `'views/'` directory to the browser, when user requests the `/checkout` path .

# Proceeding with payment

We define then the `/ProcessingPayment` route which will be requested when the user clicks the "Pay with BTC" button to generate and render the Bitcoin payment URL:

```
app.use(bodyParser.json());
app.post("/ProcessingPayment", function(req, res) {
  var amount_ = req.body.amount;
  var resp = compose_uri(amount_)+"?amount="+amount_;
  res.send(resp);
});
```

# Requesting payment details

As defined by the protocol, when the customer clicks on the payment link, its wallet (bitcoin client) will request the `/request` endpoint to get the merchant's payment request with all the necessary data.

```
var urlencodedParser = bodyParser.urlencoded({ extended: false });
app.get("/request", urlencodedParser, function(req, res) {
  var amount = req.query.amount;
  amount = (amount === undefined) ? 0 : amount; // set amount to 0
  if undefined
    var merchant_outputs = []; // Where payment should be sent
    var outputs = new PaymentProtocol().makeOutput();
    outputs.set('amount', amount);
    var script =
    bitcore_lib.Script.buildPublicKeyHashOut(Merchant_address.toString());
    outputs.set('script', script.toBuffer());
    merchant_outputs.push(outputs.message);
});
```

Here the merchant's server constructs a P2PKH transaction with the amount handed from the client side. Then within the same route we wrap inside a `PaymentRequest` message with relevant details about the payment as following:

```
var details = new PaymentProtocol().makePaymentDetails();
var now = Date.now() / 1000 | 0;
details.set('network', 'test');
details.set('outputs', merchant_outputs);
details.set('time', now); //Unix timestamp when the PaymentRequest was
created.
details.set('expires', now + 60 * 60 * 24); //timestamp after which the
PaymentRequest should be considered invalid.
details.set('memo', 'A payment request from the merchant. ');
details.set('payment_url', "http://" + IP + ":" + http_port + "/payment?
id=12345"); //location where a Payment message may be sent to obtain a
```

```
PaymentACK.  
details.set('merchant_data', new Buffer("Transaction N 12345"));  
//identify the payment request
```

---

For more information about these fields, you can visit the official [BIP70 documentation](#). After defining the payment request details we form the final request as following :

```
var request = new PaymentProtocol().makePaymentRequest();  
request.set('payment_details_version', 1);  
var certificates = new PaymentProtocol().makeX509Certificates();  
certificates.set('certificate', dcert);  
request.set('pki_type', 'x509+sha256');  
request.set('pki_data', certificates.serialize());  
request.set('serialized_payment_details', details.serialize());  
request.sign(mkey);  
var rawbody = request.serialize(); // serialize the request  
res.set({  
  'Content-Type': PaymentProtocol.PAYMENT_REQUEST_CONTENT_TYPE,  
  'Content-Length': request.length,  
  'Content-Transfer-Encoding': 'binary'  
});
```

---

A `PaymentRequest` is optionally tied to a merchant's identity using public-key infrastructure (PKI) specified in `pki_type`. We sign it using the private key that corresponds to the public key in `pki_data` before forwarding it to the client side. The response format changes depending on the requester whether a bitcoin client or a browser :

```
if (req.query.browser==1) {  
  var buf = new Buffer(rawbody, 'binary').toString('base64');  
  res.contentType(PaymentProtocol.PAYMENT_REQUEST_CONTENT_TYPE);  
  res.send(buf);  
} else {  
  //response for bitcoin core client  
  res.status(200).send(rawbody);  
}
```

---

# Receiving and acknowledging payment

Once the customer verifies the payment request, he posts a payment transaction to the merchant server (toward `payment_url` which points to the `payment/` route). Therefore, in this route we extract the payment details (payment message) from the client's transaction as follows:

```
var rawBodyParser = bodyParser.raw({type:
PaymentProtocol.PAYMENT_CONTENT_TYPE});
app.post("/payment", rawBodyParser, function(req, res) {
    var body = PaymentProtocol.Payment.decode(req.body);
    var payment = new PaymentProtocol().makePayment(body);
    var refund_to = payment.get('refund_to'); //output where a refund
should be sent.
    var memo = payment.get('memo');
    var Rawtransaction = payment.get('transactions')
[0].toBuffer();/*One or more valid, signed Bitcoin transactions that
fully pay the PaymentRequest*/
    var TransactionToBroadcast = new
bitcore_lib.Transaction(Rawtransaction).toString('hex');
/* potentially broadcast the transaction as we did in the first chapter
using chain.so/api/ */
});
```

Depending on your design you can choose which side will broadcast the payment to the bitcoin network, whether the server or the customer's wallet. Still, in the second section ([Bitcoinj](#)) we will instead request the customer's approval and then let the merchant forward the transaction to the network using a web API ( or by using [bitcore-p2p](#)).

The final action in the payment process is sending to the client a payment acknowledgement message with a receipt ID. It's possible

you can listen on the bitcoin network for whether the transaction took place before sending such a message.

```
var ack = new PaymentProtocol().makePaymentACK();
ack.set('payment', payment.message);
ack.set('memo', 'Payment processed, Thank you ;) \n invoice ID
:'+req.query.id);
//store invoice details in database
var rawack = ack.serialize();
res.set({
  'Content-Type': PaymentProtocol.PAYMENT_ACK_CONTENT_TYPE,
  'Content-Length': rawack.length,
});
res.send(rawack);
```

# Invoicing

Finally, the system would be complete if it enabled the merchant to invoice the clients. For this purpose, you can add to your system a database-based invoice system to store the purchase invoices for future access. We can define an `/invoice` handler to provide the invoice details to the requester:

```
app.get("/invoice", urlencodedParser, function(req, res) {  
    var invoice_id = req.query.id;  
    var detail="details about the invoice N:"+invoice_id;  
    /*....invoice Database access..*/  
    res.send(detail);  
});
```

Now as the server is ready we need to proceed to build the front-end part.



# Client side

As for the backend, we'll use bitcore in the client-side, therefore we need to install the front-end component. For that create a bower.json file in `views/` folder with the content :

```
{
  "dependencies": {
    "bitcore-lib": "^0.15.0",
    "bitcore-payment-protocol": "1.2.2"
  }
}
```

Then run (inside the `views/` folder):

```
bower install
```

Next, we need to install a QR code library in the `views/` directory:

```
git clone https://github.com/davidshimjs/qrcodejs
```

Then create two files: `index.html` and `main.js` files in the `views/` directory. In the first of these files paste the following code:

```
<html>
<head>
  <script src="bower_components/bitcore-lib/bitcore-lib.js"></script>
  <script src="bower_components/bitcore-payment-protocol/bitcore-
payment-protocol.min.js"> </script>
  <script src="qrcodejs/qrcode.js"></script>
  <script
src="//ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"
type="text/javascript"></script>
```

```
<link rel="stylesheet" type="text/css" href="style.css" />
</head>
<body>
  <div class="main_div">
    <form id="myForm">
      
      <br>
      <h1>MVMT WATCH</h1>
      <strong>Item Details:</strong> WATER RESISTANT
      <br>
      <strong>Price :</strong> 0.888888 BC.
      <input type="hidden" id="amount" value=888888>
      <br>
      <br>
      <input
        type="submit"
        value="Pay with BTC"
        id="submit"
        onclick="event.preventDefault();ProcessingPayment()" />
    </form>
  </div>
  <script src="./main.js"></script>
</body>
</html>
```

This code builds a demo webpage with a single product (a watch) and a payment button. On the other hand, in `main.js`, we define the front end functions to interact with the payment server.

First, we define the `ProcessingPayment()` function which initiates an Ajax call to request the payment URI.

```
function ProcessingPayment() {
  var amount_ = $('#amount').val();
  $.ajax({
    method: 'POST',
    url: '/ProcessingPayment',
    data: JSON.stringify({'amount' : amount_}),
    contentType: 'application/json',
    processData: false,
    success: function(data) {
      pay(data);
    }
  })
}
```

```
});  
}
```

The server will answer back with a payment link which will be displayed as a URL and QR code using the method `pay()` :

```
function pay(pay_url) {  
    document.write("<body><div class='pay_div'><h1>Quick Checkout</h1>  
<div class='result' id='result' name='result'> <div class='overview'>  
Payment URL : <a href=" +pay_url+ ">" + pay_url + "</a> </div><br> <div  
id='qrcode'></div> <input type='hidden' id='amount' value='888888'>  
<br> <input type='button' value='Transaction Details'  
onclick='check_details()' id='check' class='check'><div  
class='details'></div></div><script src='./main.js'></script> <link  
rel='stylesheet' type='text/css' href='style.css' /></body>");  
  
    var qrcode = new QRCode(document.getElementById("qrcode"), {  
        text: pay_url.toString(),  
        width: 128,  
        height: 128,  
        colorDark : "#000000",  
        colorLight : "#ffffff",  
        correctLevel : QRCode.CorrectLevel.H  
    });  
}
```

We define then a `check_details()` method to request from the server the payment details when the customer presses the "transaction details" button.

```
function check_details() {  
    var amount_ = $('#amount').val();  
    $.ajax({  
        method: 'GET',  
        url: '/request?amount='+amount_+'&browser=1',  
        datatype: 'binary',  
        processData: false,  
        success: function(data) {  
            get_payment_details(data);  
        }  
    });  
}
```

In the last step, once the payment request details are received they will be unpacked and displayed using the following `get_payment_details` method :

```
function get_payment_details(rawbody) {
  try {
    var body = PaymentProtocol.PaymentRequest.decode(rawbody);
    var request = (new PaymentProtocol()).makePaymentRequest(body);
    var version = request.get('payment_details_version');
    var pki_type = request.get('pki_type');
    var pki_data = request.get('pki_data');
    var serializedDetails =
request.get('serialized_payment_details');
    var signature = request.get('signature');
    var verified = request.verify();
    verified=(verified) ? "Valid" : verified;
    var decodedDetails =
PaymentProtocol.PaymentDetails.decode(serializedDetails);
    var details = new
PaymentProtocol().makePaymentDetails(decodedDetails);
    var network = details.get('network');
    var outputs = details.get('outputs');
    var time = details.get('time');
    var expires = details.get('expires');
    var memo = details.get('memo');
    var payment_url = details.get('payment_url');
    var merchant_data = details.get('merchant_data');

    $('<div>Details</div>').append('<h2>Invoice :</h2><ul><li> Network :
'+network+'</li><li>Transaction Timestamp : '+time+'</li><li>Expiration
Date: '+expires+'</li><li>Merchant data : '+merchant_data+'</li><li>
Merchant Signature verification: '+verified+'</li><li>Memo:
'+memo+'</li><li> Total : 0.00888888</li>');
  } catch (e) {
    console.log(('Could not parse payment protocol: ' + e));
  }
}
```

The displayed details are very important, especially the validation of the merchant identity status using `request.verify()` which validates the payment request signature against the merchant's identity.

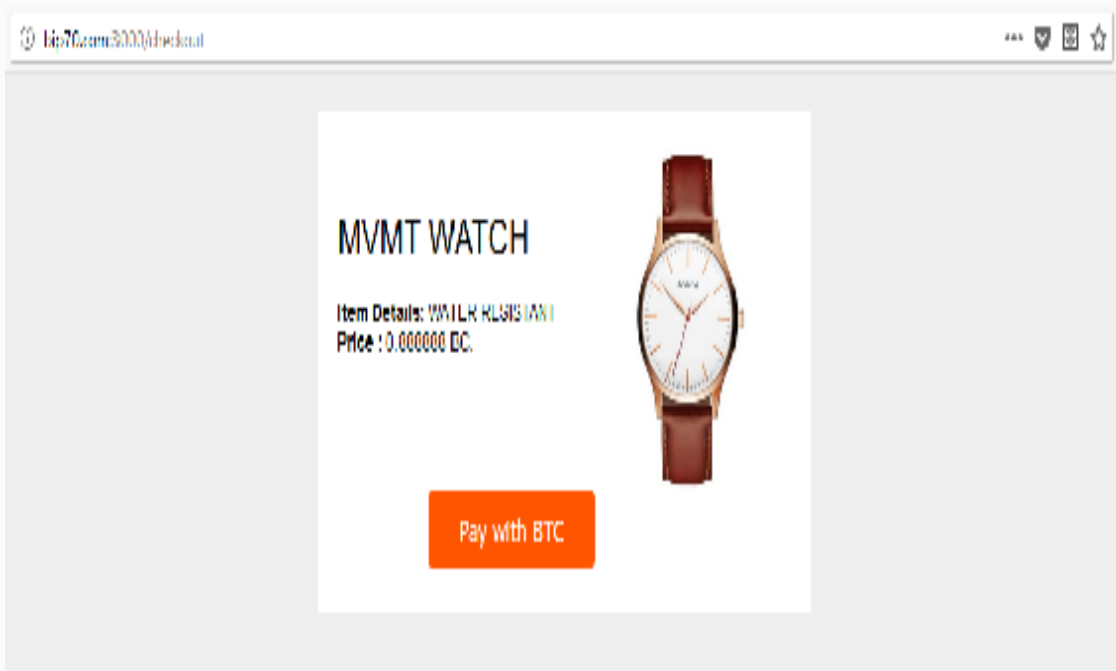
*Instead of building the front end from scratch we can use a scaffolding tool such [Yoman generator](#) to help you generate new web project structure in a matter of seconds*

Great, now the application is ready to be tested. Let's check it out.

# Preview your application

It's time to try what we have built. We need first to start the merchant server by running `npm start` from the project root ('bitcoin\_payment/'). Then, open your browser to visit the URL :  
`http://<Machine_IP>:3000/checkout`

If all went well, the server will serve you a store page with a single item and "Pay with bitcoin" button as shown the following picture:



*While I am testing on the same Linux machine, I have changed the domain name for localhost to bip70.com by editing /etc/hosts file and adding 127.0.0.1 bip70.com*

Once the client chooses to pay with Bitcoin, he will be redirected to a new view with a custom payment URL :

## Quick Checkout

Payment URL : <bitcoin:mhc5YipxN6GhRRXtgakRBjrNUCbz6ypg66?amount=0.00888888&message=payment%20request&r=http://192.168.1.6:3000/request?amount=888888>

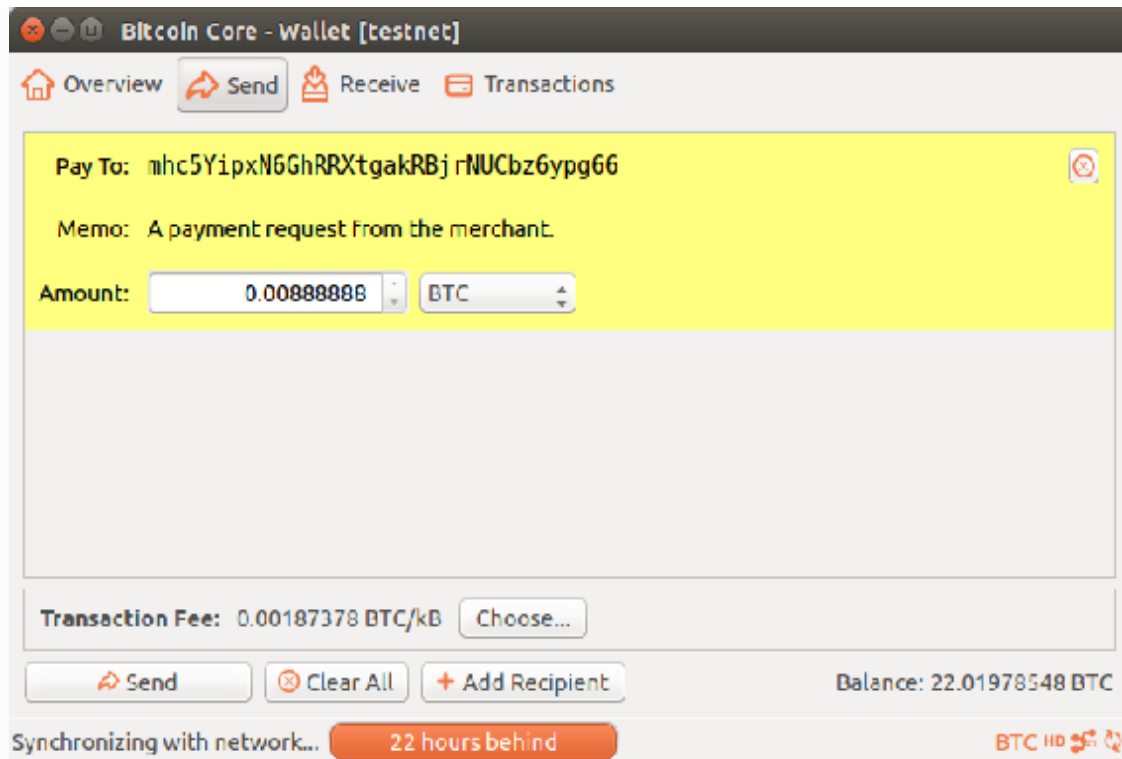


Transaction Details

### Invoice :

- Network : test
- Transaction Timestamp : 1534338057
- Expiration : 1534424457
- Merchant data : Transaction N 12345
- Merchant Signature verification: Valid
- Memo: A payment request from the merchant.
- Total : 0.0088888

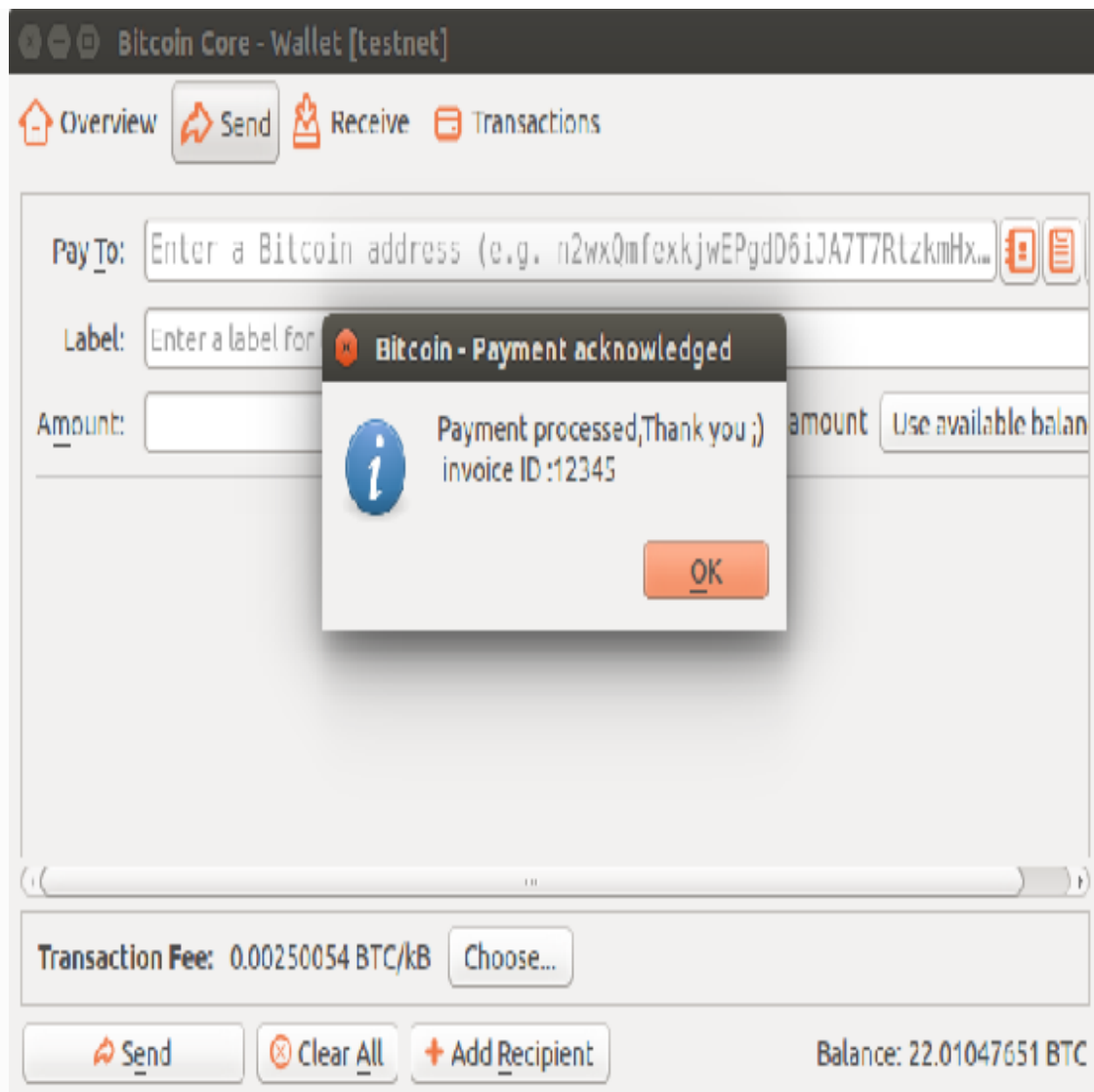
The customer can check the transaction details before proceeding with the payment by pressing the `Transaction details` button. After the customer clicks on the payment link the browser will open the bitcoin client after asking for the authorization.



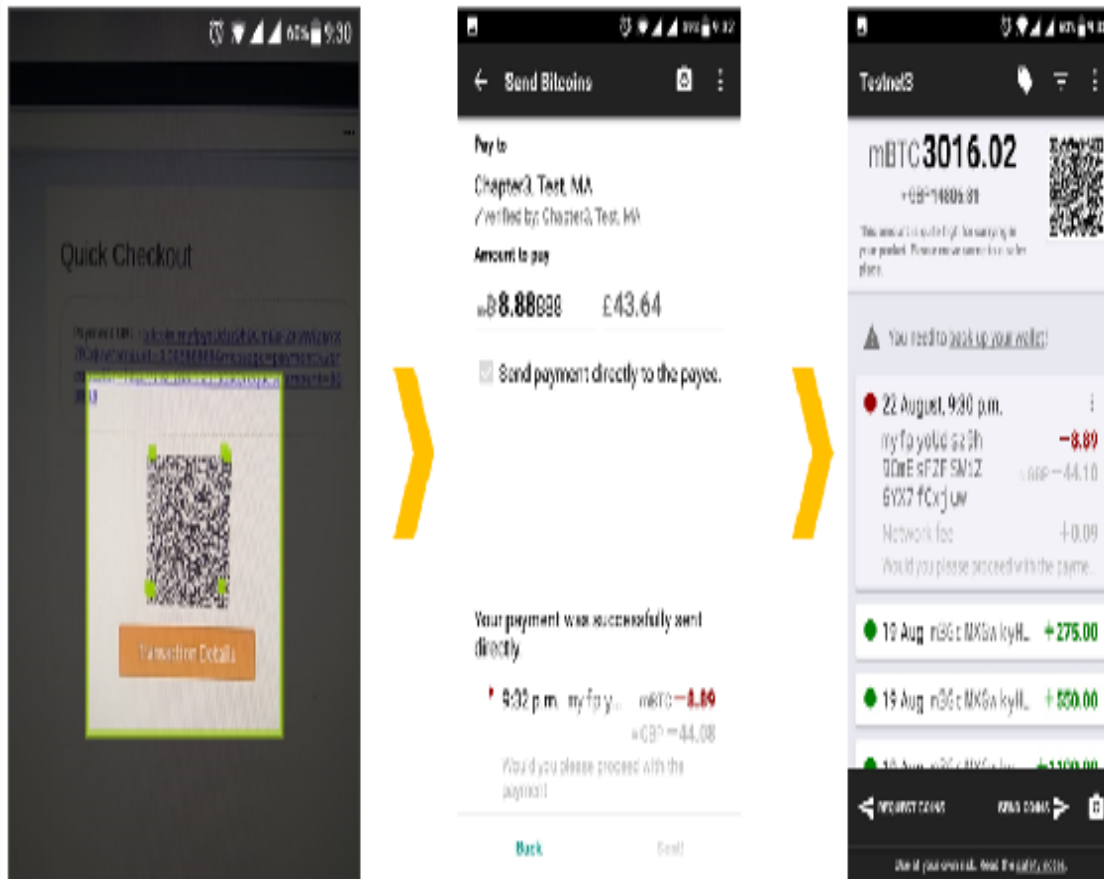
Once loaded, the Bitcoin client connects to the payment server and gets the payment details (amount, merchant address, etc...) and prepares a payment transaction. We can see the memo sent from the merchant server displayed in the client interface. After examining the amount (which he can't edit), the client will approve the transaction and send it directly to the Bitcoin network.

When the transaction is sent by the client, the Bitcoin client will display the payment acknowledgement message sent from the server confirming the payment as shown in the following image.





On the other hand the QR code helps users to pay using their smartphone. To test this ability you can install a Bitcoin wallet from Google Play store such as Testnet Wallet or copay and make sure you're connected to the same network. When you scan the QR code, you'll encounter an unavoidable error indicating that the certificate isn't signed by a trusted authority. Thus, we will need to add the merchant's certificate to the mobile OS. Once done, you can enjoy trying the payment process using the Bitcoin mobile wallet as shown below:



At this point we have successfully built an online web store which accepts bitcoin payments. To complete the experience, let's build a light Java Bitcoin wallet to pay with instead of relying on the installed Bitcoin clients.

# Payment protocol using BITCOINJ

If you're a Java Developer who would like to get started with building bitcoin applications, this part will be your best starting point. We will build a Java client that implements the payment protocol (BIP 70) using BitcoinJ, and interact with our payment NodeJs server.

# Prerequisites

The first thing you should do to follow this guide is to setup your own Java development environment. This walkthrough assumes that the latest version of following elements is installed and running on your platform of choice

- Java 8
- Eclipse IDE from <https://www.eclipse.org/downloads>

You can refer to their official documentation to properly download and install them.

# BitcoinJ installation

For this section we will use `BitcoinJ` which is a Java library designed to interact with the Bitcoin network with support for BIP70. In order to make things simpler we will install Maven's BitcoinJ plugin (version 0.14.17). If you're not a Java developer, Maven is a build automation tool for Java projects, used to download the dependencies required for a project.

Open Eclipse and create a new Maven project via (File ► New ► Other... ► Maven ► Maven) and follow the wizard instructions to finish the creation process. (Further instructions about setting up a Maven project are presented in chapter 6.) Once the project is created, browse to the `pom.xml` file and add a reference to a `BitcoinJ` and `slf4j-api` dependencies using the following lines inside:

```
<dependency>
  <groupId>org.bitcoinj</groupId>
  <artifactId>bitcoinj-core</artifactId>
  <version>0.14.7</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.21</version>
</dependency>
```

Slf4J is the logging framework used by BitcoinJ. It serves to handle the logs and help us understanding via valuable logging messages (network connection and transaction information), the behavior of the

BitcoinJ application and evaluate its execution. After that, right-click the `pom.xml` file and select `Run As ► Maven` to build and run the empty project.

Now we can finally start programming using BitcoinJ.

# BitcoinJ client

First off, in your Maven project add a new class to your `src/main/java` folder with the name `cbip70`. This class should have a main method, which will be used to communicate with the server.

# Bitcoin wallet configuration

In the first line in our code we activate BitcoinJ's logging using a Java logging formatter that writes more compact output than the default.

```
BriefLogFormatter.init();
```

Then we indicate which Bitcoin network we want to use. In this scenario, we choose the testnet as our NodeJs server is already running on testnet network by default :

```
final static NetworkParameters params = TestNet3Params.get();
```

The other available options are: `MainNetParams` and `RegTestParams`

Next, we start by initializing (inside the `main` function) a `WalletAppKit` object, to create a lightweight `SPV` (Simplified Payment Verification) BitcoinJ wallet. As mentioned in the BitcoinJ documentation, the `WalletAppKit` class wraps the boilerplate (Peers, `BlockChain`, `BlockStorage`, `Wallet`) needed to set up a new SPV BitcoinJ app.

```
WalletAppKit kit = new WalletAppKit(params, new File("."),  
"walletappkit");
```

As a result, two files are also created locally – `.wallet` (wallet ) and `.spvchain` (blockchain information) – with the specified prefix (the third argument) and stored in the indicated directory (the project root folder). Once created, we download the blockchain and wait until it's done.



```
kit.startAsync();  
kit.awaitRunning();
```

In this case, the `kit` will behave as a bitcoin node connecting to other nodes and syncing with them the blockchain (downloading only headers).

*You can use `kit.connectToLocalHost()`; in case you're using Regtest mode to connect the kit to your local Bitcoin client. Have a look at the `walletAppKit` class to learn about the available functions and understand what's happening behind the scenes.*

We generate then a Bitcoin address for the client and print it along with its balance:

```
Address CustomerAddress=kit.wallet().currentReceiveAddress();  
System.out.println("Customer's address : " + CustomerAddress);  
System.out.println("Customer's Balance : "+kit.wallet().getBalance());
```

The wallet is now ready to run but we need to send a few Bitcoins from a faucet source to the Bitcoin address returned by the `currentReceiveAddress` function. If you run the current code you'll get output similar to the following about blockchain's synchronization and the account details in Eclipse's log Viewer :



```
terminated> Chip20 [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Aug 14, 2018, 2:02:09 PM)  
[PeerGroup:INFO] INFO org.bitcoin.net.NioClientManager - Waiting 30000 ms before next connect attempt to [195.154.163.75]:18333  
[NioClientManager] INFO org.bitcoin.net.NioClientManager - Connected to testnet-seed.bitcoin.petertodd.org/82.202.297.182:18333  
[NioClientManager] INFO org.bitcoin.core.Peer - Announcing to testnet seed.bitcoin.petertodd.org/82.202.297.182:18333 as: /bitcoin:0.  
[NioClientManager] INFO org.bitcoin.net.NioClientManager - Connected to testnet seed.bitcoin.jonasschnelli.ch/18.222.184.185:18333  
[NioClientManager] INFO org.bitcoin.core.Peer - Announcing to testnet-seed.bitcoin.jonasschnelli.ch/18.222.184.185:18333 as: /bitcoin:  
[NioClientManager] INFO org.bitcoin.core.Peer - [195.154.163.75]:18333: Out version=20015, subVer=/Satoshi:0.15.1/, services=0x0, li  
[NioClientManager] INFO org.bitcoin.core.PeerGroup - [195.154.163.75]:18333: New peer (1 connected, 0 pending, 12 ms)  
[NioClientManager] INFO org.bitcoin.core.PeerGroup - Setting download peer: [195.154.163.75]:18333  
Customer's address : n3UkyjXiy86dCht5nNw0StJK2AAJ32lod0  
Customer's Balance : 165000000  
[NioClientManager] INFO org.bitcoin.net.NioClientManager - Connected to testnet-seed.bluewin.ch/190.251.103.19:18333  
[NioClientManager] INFO org.bitcoin.core.Peer - Announcing to testnet-seed.bluewin.ch/190.251.103.19:18333 as: /bitcoin:0.14.0/  
[NioClientManager] INFO org.bitcoin.Serp256k1Context - java.lang.UnsatisfiedLinkError: no serp256k1 in java.library.path  
WalletAppKit STOPPING! INFO org.bitcoin.core.PeerGroup - Awaiting PeerGroup shutdown ...
```

# Requesting payment

Now that our wallet is loaded and synced with the network, we can read the BIP70 URI provided by the merchant in order to make the payment. You can write a few lines of code to directly parse the Bitcoin payment URL and register your app as the Bitcoin URI (:bitcoin) handler. But to keep it simple, we will manually copy the payment link provided by NodeJs earlier and assign it to a local variable (I've omitted irrelevant parameters):

```
String url = "bitcoin:mhc5YipxN6GhRRXtgakRBjrNUCbz6ypg66?  
r=http://bip70.com:3000/request?amount=888888";
```

The important part in the URL is the "r" parameter which represents the merchant server, so edit it to set your server's IP or domain name.

Prior to requesting payment details from merchant's server details we can add a sanity check to evaluate if the address has enough Bitcoins:

```
if (Float.parseFloat(String.valueOf(kit.wallet().getBalance())) == 0.0)  
{  
    System.out.println("Please send some testnet Bitcoins to your  
address "+kit.wallet().currentReceiveAddress());  
} else {  
    sendPaymentRequest(url, kit);  
}
```

Then we define `sendPaymentRequest()` as following :

```
private static void sendPaymentRequest(String location, WalletAppKit k)  
{  
    try {
```

```

        if (location.startsWith("bitcoin")) {
            BitcoinURI paymentRequestURI = new BitcoinURI(location);
            ListenableFuture<PaymentSession> future =
PaymentSession.createFromBitcoinUri(paymentRequestURI, true);
            PaymentSession session = future.get();

            if (session.isExpired()) {
                log.warn("request expired!");
            } else { //payment requests persist only for a certain
duration.
                send(session, k);
                System.exit(1);
            }
        } else {
            log.info("Try to open the payment request as a file");
        }
    } catch (Exception e) {
        System.err.println( e.getMessage());
    }
}

```

This method is comprised of two steps – parsing the Bitcoin URI in order to request payment details from the specified URL, and running another function `send()` which proceeds with the payment.

The `createFromBitcoinUri` method initializes a `PaymentSession` using the payment URI. If this function is called with a second parameter set to `true`, the system trust store will be used to verify the signature provided by the payment request and a `CertPathValidatorException` exception is thrown in the failure case. `future.get()` parses the payment request which is returned as a protocol buffer.

Once the payment session is established we call the `send()` method to proceed with the payment. Note that you'll have to handle a few different exceptions, but I've set here a global try/catch for all expected exceptions to make the code cleaner for the reader.

# Sending a payment

The next function is the one to pay the bill. Before sending the Bitcoins, this function will check the payment request details including the merchant's x509 certificate. We print out in the console the payment request details to let the client know to whom he is going to pay.

```
private static void send(PaymentSession session, WalletAppKit k) {
    log.info("Payment Request");
    log.info("Amount to Pay: " +
session.getValue().toFriendlyString());
    log.info("Date: " + session.getDate());
    // Probably indicates what your are paying for.
    log.info("Message from merchant : " + session.getMemo());
    PaymentProtocol.PkiVerificationData identity =
session.verifyPki();

    if (identity != null) {
        // Merchant identity from the certificate
        log.info("Payment requester: " + identity.displayName);
        // The issuing Certificate Authority
        log.info("Certificate authority: " +
identity.rootAuthorityName);
    }
}
```

The important point in this first part is to validate the merchant's identity and signature using the PKI system. In fact, `session.verifyPki()` checks if the merchant DER certificate containing the public key corresponding to the private key used to sign the `PaymentRequest`, is signed by a trusted root authority. We display to the customer the merchant's identity and the certifying authority.

Then we call the `getSendRequest` method to get the needed information about precisely how to send money to the merchant. Until now the transaction in the request is incomplete, we need the client to confirm the payment transaction using `completeTx(req)` which adds outputs and signed inputs according to the instructions in the request. The client indicates a refund address and a short memo to the intended destination.

```
final SendRequest request = session.getSendRequest();
k.wallet().completeTx(request);
String customerMemo = "Nice Website";
Address refundAddress = new
Address(params, "mfcjN5E6vp2NWpMvH7TM2xvTywzRtNvZWR");
ListenableFuture<PaymentProtocol.Ack> future =
    session.sendPayment(ImmutableList.of(request.tx), refundAddress,
customerMemo);

if (future != null) {
    PaymentProtocol.Ack ack = future.get();
    ...
}
```

The client creates here a transaction that fully pays the `PaymentRequest` using `completeTx`. Then we call the method `sendPayment` which does not broadcast the transaction to the Bitcoin network, but instead sends a `Payment` message after the customer has authorized payment and indicated a refund address.

More specifically, if `payment_url` is specified in the merchant's payment request, then the payment message is serialized and sent as the body of the POST request to that URL. The server will forward the payment transaction to the network.

Afterwards, the customer's wallet waits for an acknowledgement of payment from the server :

```
...
System.out.println("Memo from merchant :"+ack.getMemo());
```

```
...
```

Then we put the given transaction into the wallet's pending pool :

```
...
kit.wallet().commitTx(request.tx);
...
```

At this level we have to edit the server's code (`server.js` file) to make it able to broadcast the received transaction. For that within the route `/payment` we have to add few lines broadcasting the raw transaction using the [chain.so](https://chain.so) API as we did in the first chapter.

```
...
    var Rawtransaction = payment.get('transactions')[0].toBuffer();
    var TransactionToBroadcast = new
bitcore_lib.Transaction(Rawtransaction).toString('hex');
    var ack = new PaymentProtocol().makePaymentACK();
    ack.set('payment', payment.message);
    console.log("the merchant broadcast")

    var Sendingoptions = {
        method: 'POST',
        url: 'https://chain.so/api/v2/send_tx/BTCTEST',
        body: { tx_hex: TransactionToBroadcast },
        json: true };

    rp(Sendingoptions).then(function (response) {
        var Jresponse= JSON.stringify(response);
        ack.set('memo', 'Payment processed,Thank you ;) \ninvoice ID
: '+req.query.id+"\nTransaction Details : "+Jresponse );
        var rawack = ack.serialize();
        res.set({
            'Content-Type': PaymentProtocol.PAYMENT_ACK_CONTENT_TYPE,
            'Content-Length': rawack.length,
        });
        res.send(rawack);
    });
    ...
```

The server should normally determine whether or not the transaction satisfies the conditions of payment after broadcasting the transaction. It should also wait for confirmations to make sure that the payment is received before returning an acknowledgement message, as a transaction could fail to confirm for many reasons.

However, if the Bitcoin URI returned by the payment request doesn't contain a payment URL, we broadcast directly the signed transaction from the client side :

```
...
} else {
    Wallet.SendResult sendResult = new Wallet.SendResult();
    sendResult.tx = request.tx;
    sendResult.broadcast =
k.peerGroup().broadcastTransaction(request.tx);
    sendResult.broadcastComplete = sendResult.broadcast.future();
}
```

And with this last function we have all set to try out our simple BitcoinJ application. Make sure to properly shut down all the running services when you want to stop the kit.

```
log.info("stopping..");
kit.stopAsync();
kit.awaitTerminated();
```

Congratulations. You have finished your first BitcoinJ program and got it to run. The complete source code for this section is available is available on github: [https://github.com/bellaj/BitcoinJ\\_Bip70](https://github.com/bellaj/BitcoinJ_Bip70).

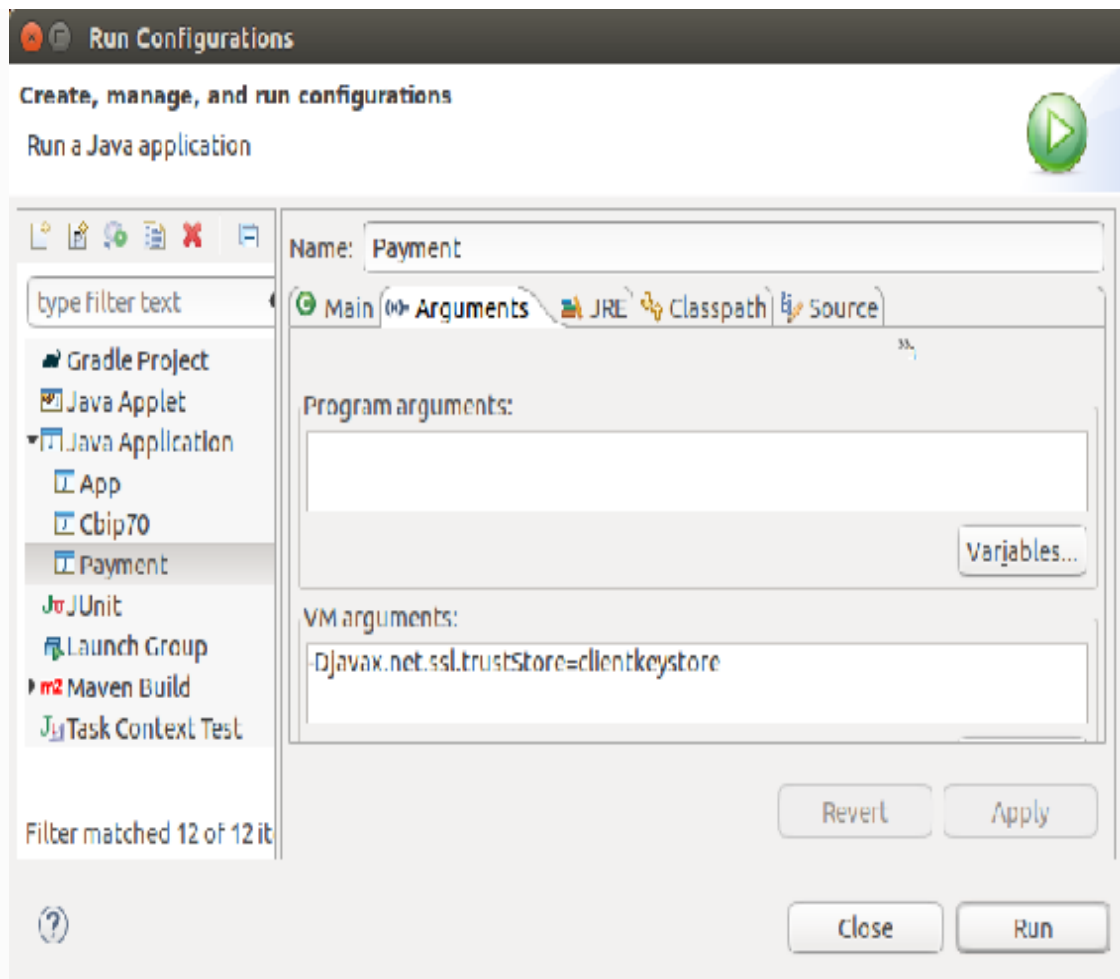
# Testing code

It's testing time but before running the project we need to define a trusted `keystore` for Java in order to validate the server's certificate, and for that we use the Java `keytool` . (Further usage details on `keytool` can be found on [oracle's documentation](#).)

```
keytool -import -keystore clientkeystore -file /path_to/cert.der -alias  
bip70.com -storepass changeit
```

Then import the resulting `clientkeystore` file into the Eclipse project. Afterwards, click "Run as" then "Run Configurations" to define the parameter `-Djavax.net.ssl.trustStore=clientkeystore` in the Arguments tab in the VM Arguments box:





*A good alternative way to use a custom trust store is to use `createFromBitcoinUri(BitcoinURI uri, boolean verifyPki, TrustStoreLoader trustStoreLoader)` where the third argument is a `TrustStoreLoader` which loads the local `KeyStore`. When it's not defined the system default trust store is used.*

When you finish, compile and run your code. If everything runs correctly you will be able to visualize in the Eclipse log viewer the SVP wallet activity, the merchant's payment request details, and the memo from the merchant when the payment is successfully proceeded, along the invoice ID.

```
[main] INFO org.bitcoinj.core.AddressResolver - Connected server: bitcoinrpc://127.0.0.1:18445
[main] INFO org.bitcoinj.core.AddressResolver - Payment request:
[main] INFO org.bitcoinj.core.AddressResolver - Amount to pay: 0.00000000 BTC
[main] INFO org.bitcoinj.core.AddressResolver - Expires: 2015-08-22 12:00:00
[main] INFO org.bitcoinj.core.AddressResolver - Message from merchant: would you please proceed with the request
[main] INFO org.bitcoinj.core.AddressResolver - Payment request: Complete, Total: 0.00000000 BTC
[main] INFO org.bitcoinj.core.AddressResolver - Constructing and sending a payment transaction
[main] INFO org.bitcoinj.core.AddressResolver - Complete: 0.00000000 BTC
[main] INFO org.bitcoinj.core.AddressResolver - Adding transaction to wallet's mempool
[main] INFO org.bitcoinj.core.AddressResolver - Transaction ID: 0000000000000000000000000000000000000000000000000000000000000000
[main] INFO org.bitcoinj.core.AddressResolver - Confirmation from the server & Transaction and Invoice ID
```

At this point, we have reached our goal of building a Java client to send a payment over the testnet network using the BIP 70 protocol. As you have witnessed, BitcoinJ is an easy-to-use and powerful framework. I would definitely recommend you spending time reading their official documentation and compiling the code samples provided there.

Finally, I would note that for the sake of brevity, the overall server and client code is intentionally unoptimized and lacks many features. You could probably go back over the code and try to include new improvements such as building a GUI. I'll be waiting for your pull requests.

# Bitcoin smart contracts

In the previous chapter we have briefly introduced the smart contract concept as self-executing programs that define a set of clauses stored on the blockchain. Although Bitcoin was the first blockchain to propose a scripting language (limited for security reasons) to control funds, the smart contract concept is usually associated with the Ethereu  
m blockchain, which provides Turing-complete smart contract languages. However, to redress the balance, many projects were initiated such as Rootstock, Counterparty and Ivy to enable building and running advanced smart contracts on Bitcoin. Welcome to Bitcoin 2.0.

# What's Rootstock

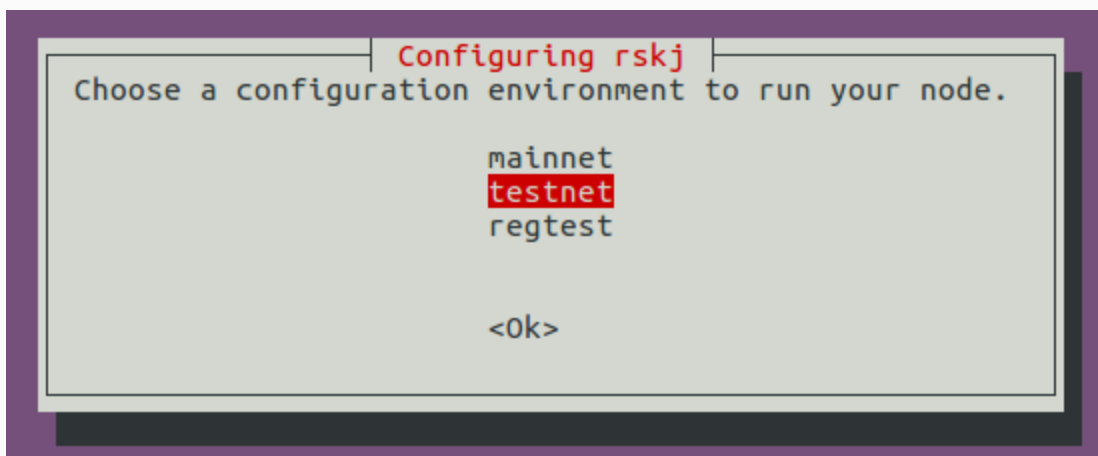
Rootstock is a project to bring smart contracts (Turing-complete smart contracts) to the ecosystem of Bitcoin. Rootstock ([www.rsk.co](http://www.rsk.co)) is a separate blockchain attached to the Bitcoin blockchain through the use of a two-way peg. When you send a Bitcoin over to Rootstock (via a special address where they will be locked and un-spensible) it becomes a "Smart" Bitcoin living in Rootstock which can be sent back to the Bitcoin chain. This concept is known as a sidechain. Technically, Rootstock nodes are using a port of Ethereum (Java source) harnessing the EVM and its Turing-complete language to write and run smart contracts. Moreover, it promises faster payment and scaling up to 100 transactions per second.

# Rootsock setup

To get started, we first need to install `rskj` (a Java implementation of the RSK protocol). The setup process is quite straightforward, and uses the following commands:

```
sudo add-apt-repository ppa:rsksmart/rskj
sudo apt-get update
sudo apt-get install rskj
```

After the installation finishes, a configuration wizard will prompt you to select your network. Let's choose testnet.



To reconfigure your node once installed, you need to change the configuration files that are located in `/etc/rsk/`. Firstly, in `node.conf` you have to enable account creation by adding `enabled = true` inside the wallet configuration section :

```
wallet {
  accounts = []
```

```
enabled = true  
}
```

---

Secondly, to change the CORS parameter from `cors = "localhost"` to `cors = ""`.

We also enable the prune service by setting the parameter `enabled = true` in the `prune` section.

# Interactions with RSK

After having configured the RSK node, we can then manage it as a service which can be started with :

```
sudo service rsk start
```

As result RSK will run as a daemon in the background and download the blockchain (a few Gbs) in the `/var/lib/rsk/database/testnet` directory. Similarly, you can stop, restart or check the situation of the service using:

```
sudo service rsk restart
sudo service rsk stop
service rsk status
```

*For a detailed overview of an RSK node's behaviour, you can consult its logs using `tail -f /var/log/rsk/rsk.log`*

The RSK project provides a useful console that enables interacting with an RSK node through RPC calls. We can install this utility using npm:

```
git clone https://github.com/rsksmart/utilities.git
cd utilities/console && npm install
```

After the installation is complete, open a new CLI and run the following command to access the console (by default RSK node runs in port 4444):

```
node console.js -server localhost:4444
```

This will open for you an interactive console that allows you to send commands to your RSK Smart node. You can find a list of available RPC commands in the official [documentation](#).

It worth mentioning that an RSK node behaves like an Ethereum node by exposing the `Web3` API over RPC, whereas the JavaScript console is a wrapper for `web3js` API (part of `Ethereum` project) to execute Web3 available calls in a Bitcoin-related environment. We will cover in depth this API in chapter 4, 5 and 6.

*To check if the blockchain is synced, we can use in the RSK console the command `web3.eth.syncing` which should return `false` (which might also mean that the node isn't syncing) and the `web3.eth.blockNumber` command should return the same block number as the latest block in RSK explorer ([explorer.testnet.rsk.co/blocks](http://explorer.testnet.rsk.co/blocks)).*

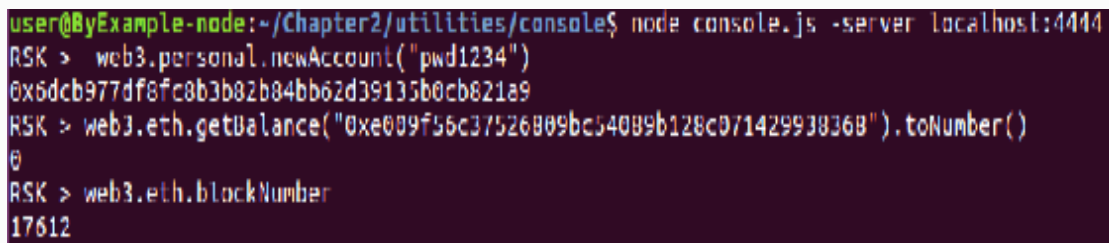


# Accounts settings

Whilst waiting for the blockchain to synchronize, let's create a new account from RSK console :

```
web3.personal.newAccount('YourPassword')  
web3.personal.unlockAccount(web3.eth.accounts[0], 'YourPassword', 0)
```

The result should look like :



```
user@ByExample-node:~/Chapter2/utilities/console$ node console.js -server localhost:4444  
RSK > web3.personal.newAccount("pwd1234")  
0x5dc977df8fc8b3b82b84bb62d39135b0cb821a9  
RSK > web3.eth.getBalance("0xe009f56c37526809bc54089b128c071429930368").toNumber()  
0  
RSK > web3.eth.blockNumber  
17612
```

Obviously the balance is null because the freshly created account hasn't yet received any SBTCs. To change that we need load this account with some free Smart Bitcoins (SBTC). For that, we can request them from an online faucet provider such as <http://faucet.testnet.rsk.co>.

You can check your new balance a few moments later using the following command :

```
web3.eth.getBalance(web3.eth.accounts[0]).toNumber()
```

Where `web3.eth.accounts[index]` is an array of the created accounts.

Note that while the blockchain isn't synced you'll get also a null balance. You can instead check the balance online using the Testnet

RSK explorer : <https://explorer.testnet.rsk.co/address/<your address>>

*Note: If you want to use the sidechain aspect of rootstock ( 2-way peg mechanism) whereby each Bitcoin can be converted into a Smart Bitcoin you should first whitelist your address (<https://github.com/rskismart/rskj/wiki/whitelisting-in-RSK>) and send the Bitcoins to the TestNet Federation address : 2MyqxrSnE3GbPM6KweFnMUqtnrzGkhT836. Then you'll need to convert your private key to an RSK private key using <https://utils.rsk.co> and import it into RSK's node.conf*

# Writing your first Bitcoin smart contract

To write smart contracts in RSK we should use Solidity, which is an Ethereum smart contract language. In this chapter we will not go into detail about Solidity but rather we will limit the explanation to how to deploy and interact with a Solidity smart contract in RSK. We will delve into Solidity in the Ethereum chapters.

As a first smart contract we consider the code snippet:

```
pragma solidity ^0.4.21;

contract store {
    string public Message;

    function set (string NewMessage) public {
        Message = NewMessage;
    }
}
```

If you're unfamiliar with Solidity, know that this smart contract defines a variable called `Message` and a setter method to edit its value. In other terms we read and store a message into the blockchain.

# Deploying the contract

If your RSK node is synchronized with the network, we can easily deploy our contract using Remix. Remix is a browser-based IDE that enables developers to compile and deploy Solidity smart contracts into the blockchain. Further instructions and details about Remix will be presented in the chapter 4 and 5.

We need to connect Remix to our running RSK node by following these steps :

- Open the Remix IDE by visiting: <http://remix.ethereum.org>
- In the *Run* tab (right menu), select Environment -> *Web3 provider*.
- In the pop up define `http://localhost:4444` as your Web3 provider endpoint.

Once connected, paste the contract code into Remix's code section and in the right-hand panel press the `Deploy` button to send a transaction that deploys the selected contract. RSK VM is based on Ethereum VM, therefore the cost of deploying and interacting with the smart contract is calculated in gas. Nevertheless the gas will be **paid in SBTC**.

While transferring the contract to the RSK network, Remix will display important information in the console located on the bottom of the main section.



# Interacting with the contract

After we've deployed the contract, we can interact with the contract instance in the Testnet network, using the same tool `Remix` we used in the deployment. If the deployment succeeded, Remix will provide you with a form in the right-hand pane to execute the contract's methods. In our case we'll pass a new message as a string (with quotes) `"hello"` to the method `set()` .

Then press the blue `Message` button to make Remix request the variable value from the contract's storage. It will return the last stored value, "hello", as expected.

Great, you've successfully built and deployed your first contract into the RSK Testnet network. If you are tempted to deploy your contract in the main network you have to whitelist your account by making a request to the RSK team, as the network is closed to whitelisted participants.

RSK isn't the only attempt to bring advanced smart contract capabilities to Bitcoin. Other projects such as [counterparty](#) aim to

provide a protocol based on Bitcoin to enable Ethereum-like smart contracts and building DApps. On the other hand, unlike the former which is based on Ethereum, Ivy is a new smart contract language that compiles to Bitcoin Script. To play around with Ivy you can use an online contracting tool <https://ivy-lang.org/bitcoin>.



# Summary

Congratulations! You are now qualified to write production applications using the most popular blockchain – Bitcoin – whether as a payment system or as a platform for running advanced smart contracts.

In this walkthrough we have consolidated our understanding of Bitcoin and dissected in depth its underlying scripting mechanism. More importantly, you know now how to integrate Bitcoin payments into applications and websites using the bitcoind library. Lastly, we discovered how to use an RSK network for deploying and interacting with Solidity smart contracts in a Bitcoin environment.

Stay tuned for the next chapter, where we will learn something very exciting. You will discover how to build your own cryptocurrency using Bitcoin's codebase. For sure it will be a golden occasion for you to deepen your understanding of Bitcoin's protocol and blockchain in general.

## **Chapter 3 : Building your own cryptocurrency**

# Introduction

In the previous chapter you learned about major concepts behind Bitcoin by building a payment system based on the Bitcoin payment protocol. Alongside this you took earlier advantage of projects like `Rootstock` to enable advanced smart contract capabilities. However, there is no better way to cement and deepen your understanding of the blockchain than to build your own cryptocurrency. Although building a currency from scratch is beyond the scope of this book, you will examine instead how to build your own cryptocurrency based on the existing Bitcoin codebase.

In this chapter we'll cover the basic concepts necessary to build an alternative cryptocurrency (known as an altcoin) that offers almost all the functionalities of Bitcoin, which we will call Readercoin. I don't expect the reader to have strong programming skills, neither deep knowledge of Bitcoin's details, consequently I present herein a simplified guide to help anyone to customize Bitcoin and create a new clone – a new cryptocurrency.

This chapter is broken into two main parts:

1. Compiling the Bitcoin code.

## 2. Designing and building Readercoin.

By the end of this this two-part guide you should be able, among other things, to clone Bitcoin, and build and mine your new currency.

# Compiling Bitcoin from source

The Bitcoin project is open sourced under the MIT license. Therefore it's possible for you to experiment with its source code and build your own derivative cryptocurrency. As a first step toward creating your own currency, it would be opportune to start by compiling the original Bitcoin source code without modification to get accustomed with the building process.

# Preparing your build system

To get the most out of this first section you'll need the following elements :

- Two or more computers or virtual machines – I will be using my laptop and a ubuntu 16.04 VM. The guide should work on 14.04 and 17.04 as well
- Text Editing Software – I'm using nano.
- Time, Patience, and lots of Arabic coffee...

Preferably, you should have some basic knowledge of C++ programming, at least you should be able to understand the basic compiling errors and how to fix them. Furthermore, this chapter covers some technical topics about the Bitcoin protocol, which you might want to read up on beforehand in the previous chapters along with the [official documentation](#).

# Installing dependencies

Before we advance further, I would like to note that we will be using the Bitcoin core project as it's the most complete implementation of the Bitcoin protocol. The project is based on many external libraries such as :

- libssl : the portion of OpenSSL which supports TLS.
- Boost C++: a set of libraries that provide support for tasks and structures such as multithreading, filesystem operations, pseudorandom number generation, etc.
- libevent: an event notification library.
- Miniupnpc: UPnP IGD client (Firewall-jumping support).
- libdb4.8 : library for the Berkeley database which is used for wallet storage (wallet.dat file).
- Qt : Qt SDK (only needed when GUI enabled)
- Protobuf: Data interchange format used for payment protocol (only needed when GUI enabled)

To install these dependencies, make sure you upgrade and update any outdated packages:

```
sudo apt-get update  
sudo apt-get upgrade
```



Instead of installing all dependencies in one shot, let's install by grouping the packages:

```
sudo apt-get install build-essential libtool autotools-dev automake  
pkg-config libssl-dev libevent-dev bsdmainutils python3 git
```

You need to carefully watch the output logs to detect if something goes wrong. Next step will be to install only the necessary part of the boost C++ library :

```
sudo apt-get install libboost-system-dev libboost-filesystem-dev  
libboost-chrono-dev libboost-program-options-dev libboost-test-dev  
libboost-thread-dev
```

If that doesn't work for any reason, you can install all the boost development packages with:

```
sudo apt-get install libboost-all-dev
```

Afterwards, you'll need to install the Berkley database, which is what the Bitcoin software uses for wallet storage, and to store necessary functions that make the wallet work correctly (BerkleyDB for wallet files, and LevelDB for blockchain indexes).

To Install Berkeley 4.8 libs on Ubuntu 16.04, run the following commands:

```
sudo apt-get install software-properties-common  
sudo add-apt-repository ppa:bitcoin/bitcoin  
sudo apt-get update  
sudo apt-get install libdb4.8-dev libdb4.8+-dev
```

# Bitcoin-qt: Qt5 GUI for Bitcoin

If you are interested in running the graphic interface GUI toolkit all you need to do is install the following QT5 dependencies:

```
sudo apt-get install libqt5gui5 libqt5core5a libqt5dbus5 qttools5-dev  
qttools5-dev-tools libprotobuf-dev protobuf-compiler qt5-default
```

Your build system is ready now, with all the packages installed we can start building the Bitcoin source code.

# Cloning the Bitcoin source code

Prior to building the code, start by creating a source directory called "workspace" where we clone the Bitcoin Core source repository:

```
mkdir workspace && cd
```

To download the Bitcoin Core source code, we'll use `git` to clone the latest version of its repository into the current local directory :

```
git clone https://github.com/bitcoin/bitcoin.git && cd bitcoin
```

The download process will take a few minutes. At the end you'll get a new directory called `bitcoin/` with all the Bitcoin source code files.

# Building Bitcoin Core

As the dependencies are installed we can build the code using autotools by running in succession the commands below :

```
./autogen.sh
./configure --with-gui=qt5 --enable-debug
sudo make
```

Pay attention to the space and double dash in front of both options `with-gui` and `enable-debug`.

It's common to supply some options to the `configure` command to change the final location of the executable program. You can explore the different possible options in the official documentation <https://github.com/bitcoin/bitcoin/blob/master/doc/build-unix.md>

As you notice we passed the `configure` file two arguments: `--with-gui=qt5` to build the graphical interface and `--enable-debug` to produce better debugging builds. If you want to build only a headless Bitcoin client, use `--without-gui`.

*In order to accelerate the building process you can skip running the tests by specifying the `--disable-tests` argument.*

After running the `make` command, the initial compile will take quite a long time. Think about taking a coffee.

If all the dependencies have been installed correctly, a test compilation should be achieved successfully at the end of compilation process. If you got an error somewhere, have a look at the compilation output to spot the error.

*Alternatively, you can run `qmake`, which is the qt make tool, before running `make` or you can install Qt Creator and open the `bitcoin-qt.pro` file located under `contrib` folder.*

Now that the software is built and ready to run you can optionally install it by running the command below:

```
sudo make install
```

This will add the Bitcoin commands to your PATH so that you can simply run `bitcoin-qt` or `bitcoind` to start the Bitcoin client without specifying the binary location. Keep in mind that you have to re-run `make install` each time you edit and compile your code to update your client.

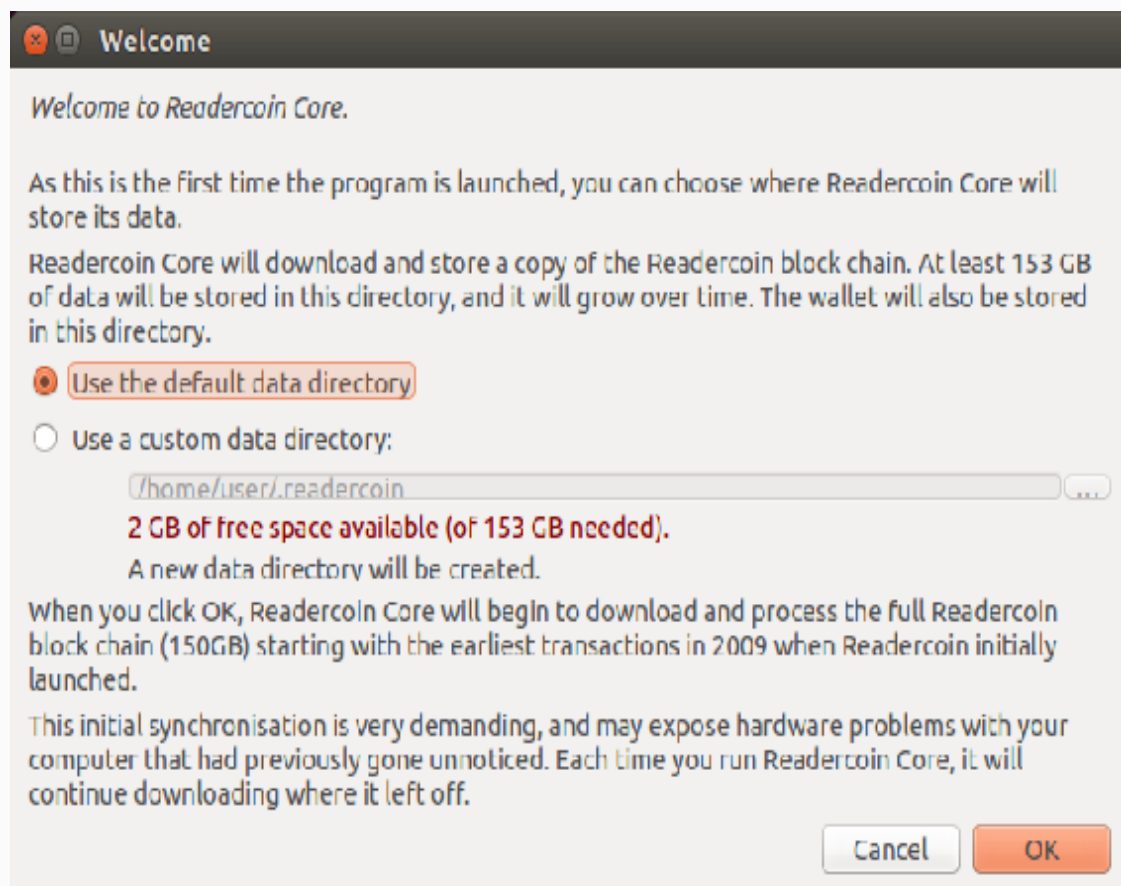
Congratulations you have compiled successfully Bitcoin from its source code.

# Checking your build worked

Here are a couple more commands to make sure that the build worked, and to make sure the executable files it was supposed to generate are actually there. From the `bitcoin/` directory run :

```
ls src/bitcoind
ls src/bitcoin-cli
ls src/qt/bitcoin-qt
```

Running this command will show you the Bitcoin Core welcome screen similar the following figure :



Excellent! If Bitcoin runs without error that means your build environment has everything you need to create an altcoin. Let's start making your altcoin.

# New cryptocurrency: ReaderCoin

Big time! Finally.

Although "Readercoin" will be a derivative coin from Bitcoin, we will design it differently. We need to define, amongst other characteristics, how the new altcoin is minted, money supply, altcoin brand, etc...

The following table summarizes the important parameters we'll define for Readercoin :

|                                |                                |
|--------------------------------|--------------------------------|
| <b>Coin's Name</b>             | <b>Readercoin</b>              |
| <b>Con's Symbol</b>            | RDC                            |
| <b>Block Time</b>              | 2.5 minute                     |
| <b>Initial block reward</b>    | 10                             |
| <b>Halving interval</b>        | Every 100000 blocks (174 days) |
| <b>Ports</b>                   | RPC port: 19333                |
|                                | P2P port: 4333                 |
| <b>Total supply</b>            | 2000000 units                  |
| <b>difficulty readjustment</b> | After 576 blocks               |
| <b>Block size</b>              | Up to 8 Mb                     |



# Cloning Bitcoin

Bitcoin is a highly active project frequently releasing new releases. The current version at the time of writing is 0.15 – you can view a list of project releases at <https://bitcoin.org/en/version-history>. From one version to another a lot of features get added or removed, therefore you must examine the changelog to see if there is any special feature you are interested in. For example, internal mining got removed in version 0.13.

To build ReaderCoin, I've opted for using the last major version, 0.15. Create a directory called `readercoin` and clone Bitcoin 0.15 code as follow:

```
git clone -b 0.15 https://github.com/bitcoin/bitcoin.git readercoin &&  
cd readercoin
```

git gives us access to all past and future versions, hence you can choose different branches using the `-b` option .

Now we have to remove Bitcoin's ties to the original Bitcoin repository. This can be easily done with the command:

```
rm -rf .git
```

To enable git versioning, you can initialize a new git repository on your newly created folder "Readercoin", add the existing files, and commit them:

```
git init
git add -A
git commit -m "initial commit"
```

Besides being a good habit, maintaining a remote repository will help us to compile our ReaderCoin client on different machines without the need to copy the code each time.

Go to the GitHub website and create a new online repository, then add your repository's git URL as the remote repository. Afterwards, push your initial commit:

```
git remote add origin https://github.com/bellaj/Readercoin_.git(use
instead YOUR NEW GITHUB REPO's url)
git push -f origin master
```

Remember before you start editing the code, that you can at any level of the following guide compile the code using `make` to check if the addition process is successfully done or not. It's advised to commit and push working changes to your remote repository as it will help you to make rollbacks if there are any compilation issues, or if you want to compile code on other machines.

# Readercoin rebranding

We kick off by rebranding the Bitcoin project into ReaderCoin by renaming the 'bitcoin' filenames to 'readercoin' using the following commands:

```
find . -exec rename 's/bitcoin/readercoin/' {} ";"  
find . -exec rename 's/btc/rdc/' {} ";"
```

These Linux commands will rename every file that has 'bitcoin' or 'btc' in the name, recursing into subdirectories as needed.

As changing filenames will affect source files and make files, breaking paths and include statements, we need to replace all the occurrences of the old Bitcoin name and abbreviation of the BTC with the new name ReaderCoin and acronym RDC recursively in the code. To be sure of replacing all occurrences we use different capitalizations:

```
find ./ -type f -readable -writable -exec sed -i  
"s/bitcoin/readercoin/g" {} ";"  
find ./ -type f -readable -writable -exec sed -i  
"s/Bitcoin/Readercoin/g" {} ";"  
find ./ -type f -readable -writable -exec sed -i  
"s/BitCoin/ReaderCoin/g" {} ";"  
find ./ -type f -readable -writable -exec sed -i  
"s/BITCOIN/READERCOIN/g" {} ";"  
find ./ -type f -readable -writable -exec sed -i  
"s/bitcoind/readercoind/g" {} ";"  
find ./ -type f -readable -writable -exec sed -i "s/BTC/RDC/g" {} ";"  
find ./ -type f -readable -writable -exec sed -i "s/btc/rdc/g" {} ";"
```

Instead of running these commands separately, you can put them in a single bash script. The command above will take a few minutes to complete. To check if there are any remaining occurrences of the Bitcoin string use: `grep -ri "bitcoin"`

*Alternatively, you can use a text editor like gedit to find and replace all Bitcoin and BTC occurrences in the project source files.*

Changing the Bitcoin occurrences will also affect the git files. Hence, If you want to keep using git you have to use the following commands to avoid git's bad index file signature error:

```
rm -f .git/index
git reset
git add -A
git commit -m "rename bitcoin into readercoin occurrences"
git push origin master
```

To start code editing, change the working directory to "src" (`cd src`).

# Changing the ports

To operate, Bitcoin uses two ports on which the client will listen to establish connections, namely:

- p2p port (by default 8333 in mainnet and 18333 in testnet)
- RPC port (by default 8332 in mainnet and 18332 in testnet)

We will define different ports for both mainnet and testnet, to be able to run your altcoin alongside a Bitcoin client without port overlap. Remember, mainnet and testnet are acronyms for main and test public networks.

Get your favorite text editor and open up `chainparams.cpp` located in the `src/` folder. Find the network parameters, including the ports, and then change the following variables :

- For mainnet (`CMainParams` class)

```
chainparams.cpp: nDefaultPort = 9333;  
chainparams.cpp: nDefaultPort = 19333;
```

- For testnet (`CTestNetParams` class):

```
chainparamsbase.cpp: nRPCPort = 9332;  
chainparamsbase.cpp: nRPCPort = 19332;
```

To make modifications consistent with the whole code, change the port occurrences in all files using Linux commands:

```
find ./ -type f -readable -writable -exec sed -i "s/8332/9332/g" {} ";"  
find ./ -type f -readable -writable -exec sed -i "s/8333/9333/g" {} ";"
```

Feel free to choose any port you like but avoid the mainstream ones (below 1000, like 80 for HTTP).

Great! Remember that you can always change either the Bitcoin port or your altcoin's port just by providing the options `-rpcport` and `-port` in the command line or in the file `readercoin.conf` without editing the Bitcoin codebase.

# Changing pchMessageStart

The Bitcoin node broadcasts in each P2P frame 4 bytes known as magic values to ensure that only the client/peers belonging to the same network can communicate. The magic bytes are represented in the array `pchMessageStart[]` to identify the used network as described in the table below.

| Network  | Magic value | Sent over wire as |
|----------|-------------|-------------------|
| main     | 0xD9B4BEF9  | F9 BE B4 D9       |
| testnet  | 0xDAB5BFFA  | FA BF B5 DA       |
| testnet3 | 0x0709110B  | 0B 11 09 07       |
| namecoin | 0xFEB4BEF9  | F9 BE B4 FE       |

To differentiate our ReaderCoin network we will change `pchMessageStart` values by incrementing them by 2. Always in `chainparams.cpp`, edit `pchMessageStart` in the `CMainParams` class using the following values:

```
pchMessageStart[0] = 0xfd;  
pchMessageStart[1] = 0xc0;  
pchMessageStart[2] = 0xb6;  
pchMessageStart[3] = 0xdb;
```

For `CTestNetParams` class:

```
pchMessageStart[0] = 0xd;  
pchMessageStart[1] = 0x13;  
pchMessageStart[2] = 0xb;  
pchMessageStart[3] = 0x9;
```

For `CRegTestParams` (represents regression test network):

```
pchMessageStart[0] = 0xfc;  
pchMessageStart[1] = 0xc1;  
pchMessageStart[2] = 0xb7;  
pchMessageStart[3] = 0xdc;
```

There is no special reason behind these specific values, you can use any value between the hexadecimal values: 0 and 0xFF



# Genesis block

Now we will deal with the crucial part: the genesis block.

A genesis block is the first block in a blockchain. When a node boots, it initializes its copy of the blockchain alongside the genesis block and then begins the synchronization process. To start a new chain for our currency, we need to forge a new genesis block and override the original one hardcoded in the Bitcoin code as it was set for older date (January 2009).

Here's the source code of the function that generates the genesis blocks defined in `chainparams.cpp`:

```

model = keras.Sequential([
    Input(shape=(128, 128, 3)),
    Conv2D(32, (3, 3), activation='relu'),
    Conv2D(32, (3, 3), activation='relu'),
    Conv2D(64, (3, 3), activation='relu'),
    Conv2D(64, (3, 3), activation='relu'),
    Conv2D(128, (3, 3), activation='relu'),
    Conv2D(128, (3, 3), activation='relu'),
    Conv2D(256, (3, 3), activation='relu'),
    Conv2D(256, (3, 3), activation='relu'),
    Conv2D(512, (3, 3), activation='relu'),
    Conv2D(512, (3, 3), activation='relu'),
    Conv2D(1024, (3, 3), activation='relu'),
    Conv2D(1024, (3, 3), activation='relu'),
    Flatten(),
    Dense(1000, activation='softmax')
])

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_data_generator.flow(train_data, train_labels, batch_size=32),
        validation_data=(validation_data, validation_labels),
        epochs=100,
        verbose=1)

```

We can easily spot some pre-defined values such as the key for the genesis coinbase transaction and a timestamp message, etc. Editing

the content of this block implies calculating a new genesis hash required for other parameters within the `chainparams.cpp` code.

For creating the new genesis block we'll use a dedicated Python script called GenesisH0.

For that, in a new terminal, clone the genesis generator script (GenesisH0) from its GitHub repository:

```
git clone https://github.com/lhartikk/GenesisH0.git && cd GenesisH0
```

Then install the required package:

```
sudo pip install scrypt construct==2.5.2
```

To reproduce the original genesis block run the GenesisH0 with the following arguments:

```
python genesis.py -z "The Times 03/Jan/2009 Chancellor on brink of  
second bailout for banks" -n 2083236893 -t 1231006505 -v 5000000000
```

You need to understand what means each argument to replace with appropriate value:

- The `-z` option indicates an optional timestamp represented by an arbitrary paraphrase with a date, usually a headline of a news article can be chosen. The Bitcoin's genesis block famously contains the dated title of an article in the Financial Times: *"The Times 03/Jan/2009 Chancellor on brink of second bailout for banks"*

This is probably intended as proof that no premining has taken place before 2009. You can put a recent news headline or any time-related

information. For example, I'll use the book's name with the publication year "blockchain by example 2018".

- For the nonce (-n) you can set any value to be used as a start nonce.
- For the epoch (-t) you can use the current time epoch. You can get the current epoch time online from: [www.epochconverter.com/](http://www.epochconverter.com/) OR you can generate it from the command line of most \*nix systems with this code: `date +%s`
- For -v you need to determine the coin reward value and multiply it with \* 100000000. For example, if you have a block reward of 50 it would be -v 5000000000. We will generate only 10 ReaderCoins for the genesis block.
- -b represents the target in compact representation, associated to a difficulty of 1. To get a block in 2.5 minutes we will use  
`0x1e0ffff0`

We end up with the following command with different arguments to generate the new genesis block:

```
python genesis.py -z "Blockchain by example 2018" -n 1 -t 1529321830 -v  
10000000000 -b 0x1e0ffff0
```

After a short while you should see an output similar to the following:

```
user@ByExample-node: ~/GenesisH0
user@ByExample-node:~/GenesisH0$ python genesis.py -z "Blockchain by example 2018" -n 1 -t 1529321830 -v 1000000000 -b 0x1e0ffff0
34ffff001d01041a426c6f636b636861696e206279206578616d706c652032303138
algorithm: SHA256
merkle hash: 6bc2585d63185acf3868cc34e0b017b3fb41c00938eb09bc52a3cf73a31ec6a8
pszTimestamp: Blockchain by example 2018
pubkey: 04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f
time: 1529321830
bits: 0x1e0ffff0
Searching for genesis hash..
genesis hash found!
nonce: 490987
genesis hash: 000001a9bbae8bb141c6941838bdacdbcf474b6ed28a0b18b2120b60a68f00ee
user@ByExample-node:~/GenesisH0$
```

Bingo, you now have the genesis block information that you need to use in your codebase.

Except the first line, which is the Scriptsig of the genesis transaction, the rest of the output results are identified by an expressive keyword.

Let's edit the `/src/chainparams.cpp` file accordingly to integrate the new generated genesis block. Specifically, our target will be the function `CreateGenesisBlock` :

```
1 // CreateGenesisBlock:
2 // Creates a new block of type tx_main_type with all txs,
3 // a Merkle tree, timestamp, a new scriptSig, and a new valid-witness.
4 // Returns the block object.
5 // The block must not have any unconfirmed transactions (see ValidateBlock)
6 // The block must not have any unconfirmed transactions (see ValidateBlock)
7 // The block must not have any unconfirmed transactions (see ValidateBlock)
8 // The block must not have any unconfirmed transactions (see ValidateBlock)
9 // The block must not have any unconfirmed transactions (see ValidateBlock)
10 // The block must not have any unconfirmed transactions (see ValidateBlock)
11 // The block must not have any unconfirmed transactions (see ValidateBlock)
12 // The block must not have any unconfirmed transactions (see ValidateBlock)
13 // The block must not have any unconfirmed transactions (see ValidateBlock)
14 // The block must not have any unconfirmed transactions (see ValidateBlock)
15 // The block must not have any unconfirmed transactions (see ValidateBlock)
16 // The block must not have any unconfirmed transactions (see ValidateBlock)
17 // The block must not have any unconfirmed transactions (see ValidateBlock)
18 // The block must not have any unconfirmed transactions (see ValidateBlock)
19 // The block must not have any unconfirmed transactions (see ValidateBlock)
20 // The block must not have any unconfirmed transactions (see ValidateBlock)
21 // The block must not have any unconfirmed transactions (see ValidateBlock)
22 // The block must not have any unconfirmed transactions (see ValidateBlock)
23 // The block must not have any unconfirmed transactions (see ValidateBlock)
24 // The block must not have any unconfirmed transactions (see ValidateBlock)
25 // The block must not have any unconfirmed transactions (see ValidateBlock)
26 // The block must not have any unconfirmed transactions (see ValidateBlock)
27 // The block must not have any unconfirmed transactions (see ValidateBlock)
28 // The block must not have any unconfirmed transactions (see ValidateBlock)
29 // The block must not have any unconfirmed transactions (see ValidateBlock)
30 // The block must not have any unconfirmed transactions (see ValidateBlock)
31 // The block must not have any unconfirmed transactions (see ValidateBlock)
32 // The block must not have any unconfirmed transactions (see ValidateBlock)
33 // The block must not have any unconfirmed transactions (see ValidateBlock)
34 // The block must not have any unconfirmed transactions (see ValidateBlock)
35 // The block must not have any unconfirmed transactions (see ValidateBlock)
36 // The block must not have any unconfirmed transactions (see ValidateBlock)
37 // The block must not have any unconfirmed transactions (see ValidateBlock)
38 // The block must not have any unconfirmed transactions (see ValidateBlock)
39 // The block must not have any unconfirmed transactions (see ValidateBlock)
40 // The block must not have any unconfirmed transactions (see ValidateBlock)
41 // The block must not have any unconfirmed transactions (see ValidateBlock)
42 // The block must not have any unconfirmed transactions (see ValidateBlock)
43 // The block must not have any unconfirmed transactions (see ValidateBlock)
44 // The block must not have any unconfirmed transactions (see ValidateBlock)
45 // The block must not have any unconfirmed transactions (see ValidateBlock)
46 // The block must not have any unconfirmed transactions (see ValidateBlock)
47 // The block must not have any unconfirmed transactions (see ValidateBlock)
48 // The block must not have any unconfirmed transactions (see ValidateBlock)
49 // The block must not have any unconfirmed transactions (see ValidateBlock)
50 // The block must not have any unconfirmed transactions (see ValidateBlock)
51 // The block must not have any unconfirmed transactions (see ValidateBlock)
52 // The block must not have any unconfirmed transactions (see ValidateBlock)
53 // The block must not have any unconfirmed transactions (see ValidateBlock)
54 // The block must not have any unconfirmed transactions (see ValidateBlock)
55 // The block must not have any unconfirmed transactions (see ValidateBlock)
56 // The block must not have any unconfirmed transactions (see ValidateBlock)
57 // The block must not have any unconfirmed transactions (see ValidateBlock)
58 // The block must not have any unconfirmed transactions (see ValidateBlock)
59 // The block must not have any unconfirmed transactions (see ValidateBlock)
60 // The block must not have any unconfirmed transactions (see ValidateBlock)
61 // The block must not have any unconfirmed transactions (see ValidateBlock)
62 // The block must not have any unconfirmed transactions (see ValidateBlock)
63 // The block must not have any unconfirmed transactions (see ValidateBlock)
64 // The block must not have any unconfirmed transactions (see ValidateBlock)
65 // The block must not have any unconfirmed transactions (see ValidateBlock)
66 // The block must not have any unconfirmed transactions (see ValidateBlock)
67 // The block must not have any unconfirmed transactions (see ValidateBlock)
68 // The block must not have any unconfirmed transactions (see ValidateBlock)
69 // The block must not have any unconfirmed transactions (see ValidateBlock)
70 // The block must not have any unconfirmed transactions (see ValidateBlock)
71 // The block must not have any unconfirmed transactions (see ValidateBlock)
72 // The block must not have any unconfirmed transactions (see ValidateBlock)
73 // The block must not have any unconfirmed transactions (see ValidateBlock)
74 // The block must not have any unconfirmed transactions (see ValidateBlock)
75 // The block must not have any unconfirmed transactions (see ValidateBlock)
76 // The block must not have any unconfirmed transactions (see ValidateBlock)
77 // The block must not have any unconfirmed transactions (see ValidateBlock)
78 // The block must not have any unconfirmed transactions (see ValidateBlock)
79 // The block must not have any unconfirmed transactions (see ValidateBlock)
80 // The block must not have any unconfirmed transactions (see ValidateBlock)
81 // The block must not have any unconfirmed transactions (see ValidateBlock)
82 // The block must not have any unconfirmed transactions (see ValidateBlock)
83 // The block must not have any unconfirmed transactions (see ValidateBlock)
84 // The block must not have any unconfirmed transactions (see ValidateBlock)
85 // The block must not have any unconfirmed transactions (see ValidateBlock)
86 // The block must not have any unconfirmed transactions (see ValidateBlock)
87 // The block must not have any unconfirmed transactions (see ValidateBlock)
88 // The block must not have any unconfirmed transactions (see ValidateBlock)
89 // The block must not have any unconfirmed transactions (see ValidateBlock)
90 // The block must not have any unconfirmed transactions (see ValidateBlock)
91 // The block must not have any unconfirmed transactions (see ValidateBlock)
92 // The block must not have any unconfirmed transactions (see ValidateBlock)
93 // The block must not have any unconfirmed transactions (see ValidateBlock)
94 // The block must not have any unconfirmed transactions (see ValidateBlock)
95 // The block must not have any unconfirmed transactions (see ValidateBlock)
96 // The block must not have any unconfirmed transactions (see ValidateBlock)
97 // The block must not have any unconfirmed transactions (see ValidateBlock)
98 // The block must not have any unconfirmed transactions (see ValidateBlock)
99 // The block must not have any unconfirmed transactions (see ValidateBlock)
100 // The block must not have any unconfirmed transactions (see ValidateBlock)
```

# New pszTimestamp

Let's change the original value defined by Satoshi, located inside the `CreateGenesisBlock` function to the new passphrase we used in GenesisH0 script:

```
const char* pszTimestamp = "Blockchain by example 2018";
```

# New nonce, epoch time, and nbits

We have to locate the following line in `chainparams.cpp` in the three classes `CMainParams`, `CTestNetParams` and `CRegTestParams`

```
genesis = CreateGenesisBlock(time, nonce, bits, 1, 10 * COIN);
```

In the order change the arguments of the `CreateGenesisBlock` function using the values computed previously by `GenesisH0` script as following:

```
genesis = CreateGenesisBlock(1529321830, 490987, 0x1e0ffff0, 1, 10 *  
COIN);
```

The last argument in the function is the initial reward (10 ReaderCoins) for generating the genesis block.

# New genesis hash

While we have edited some values in the genesis block, the block's hash provided by Satoshi is no longer valid. Therefore, we have to replace it with the new one provided by GenesisH0 script.

Under the `chainParams.cpp` file you will find several occurrences of the following assertion, which refers to the old hash value of the genesis block.

```
assert (hashGenesisBlock == uint256 ("genesis block hash"));
```

These assertions verify if the genesis block hash is congruous, otherwise code execution fails and halts further code execution. You can either comment out all these assert methods or replace the hash with the new value of your generated genesis block (by adding 0x as prefix).

If you get any errors related to these assertions later when you run your client, you can check if the hashes matched by printing out the computed genesis hash:

```
printf ("Readercoin hashGenesisBlock: % s \ n",  
consensus.hashGenesisBlock.ToString (). c_str ());
```

# Editing the Merkle root

A Merkle tree is an important piece in Bitcoin's puzzle. It's a fingerprint of the entire list of transactions, thereby enabling a user to verify whether or not a transaction has been included in the block.

In `testparams.cpp` look for the following line:

```
assert( genesis.hashMerkleRoot == uint256s('merkle hash value');
```

Replace the Merkle hash with the hash calculated by GenesisH0. To check the genesis block Merkle root you can print its hash root:

```
printf ("Readercoin hashMerkleRoot:% s \ n",  
genesis.hashMerkleRoot.ToString (). c_str ());
```

All set. We have finished defining the new genesis block.



# Removing the seed nodes

Next, we'll need to remove the DNS seeds from the code. Bitcoin uses built-in DNS seeds which are a list of host names for DNS server to discover other nodes. A Bitcoin client issues DNS requests to learn about the addresses of other peer nodes, and in response the DNS seeds return a list of IP addresses of full nodes on the Bitcoin network to assist in peer discovery.

We'll need to remove or comment out the hardcoded seed nodes in the `chainparams.cpp` file. In this file you'll find a vector of seed (`vSeeds`) to which they append a list of DNS URLs:

```
vSeeds.emplace_back("seed.bitcoin.sipa.be", true);
```

We need to comment out all occurrences of `vSeeds.push_back` and `vSeeds.emplace_back` in both `CMainParams` and `CTestNetParams`.

It's not over yet with seeds as we'll need to deal with fixed IP seeds whether by commenting out the occurrences on the following line:

```
vFixedSeeds = std::vector<SeedSpec6>(pnSeed6_main, pnSeed6_main +  
ARRAYLEN(pnSeed6_main));
```

Or by replacing this line with:

```
vFixedSeeds.clear();  
vSeeds.clear();
```

Otherwise you can include your proper DNS seeds servers. In `chainparamseeds.h` you'll find a list of IPs (wrapped inside IPv6 addresses) of nodes that can be connected to for mainnet and testnet to retrieve more IP addresses of nodes that can be connected to.

```
]static SeedSpec6 pnSeed6_main[] = {  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x02, 0xe4, 0x46, 0xc8}, 3333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x04, 0xb4, 0x1d, 0x44}, 4334},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x0c, 0x0f, 0xb4, 0x1e}, 5333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x0c, 0x12, 0x4c, 0x6e}, 4333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x05, 0x37, 0x8e, 0x67}, 5333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x05, 0x2b, 0x7c, 0x9e}, 3333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x05, 0xbd, 0x8d, 0x66}, 5333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x05, 0xe2, 0x95, 0x91}, 5333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x05, 0xc4, 0x07, 0x82}, 5333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x05, 0xe4, 0x40, 0x47}, 5333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x05, 0xf8, 0x18, 0x65}, 4334},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x05, 0xfe, 0x7c, 0x37}, 5333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x05, 0x12, 0x40, 0xe1}, 4333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x05, 0xff, 0x5a, 0x8a}, 5333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x0e, 0xc0, 0x00, 0x1b}, 2133},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x12, 0x3c, 0x03, 0x56}, 5333},  
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0xff, 0x12, 0x55, 0x23, 0x53}, 5333}
```

To setup a seed node, you just need to run a normal node for ReaderCoin and add its IP address to `chainparamseeds.h` or anticipate a list of IPs that you'll use for your nodes later.

Under `contrib/seeds` there is a Python script `generate-seeds` which will help you to generate the `pnSeed6_main` and `pnSeed6_test` arrays that are compiled into the client.

In `nodes_test.txt` and `nodes_main.txt` remove the existing IPs, set all your node IPs (one IP per line), then run:

```
~/workspace/readercoin/contrib/seeds$ python3 generate-seeds.py . >
../../src/chainparamseeds.h
```

# Checkpoints

The Bitcoin Core client has hardcoded checkpoints checking that certain specific blocks should be found at certain heights. They are regularly added in new versions of the client to avoid accepting any forks from the network prior to last checkpoint, making transactions irreversible.

As our blockchain doesn't have previous blocks you have to disable these checkpoints for ReaderCoin, otherwise your node will not be able to construct additional blocks nor start mining as it will be waiting for nonexistent blocks.

In the `chainparams.cpp` file, locate the `checkpointData` map:

```
checkpointData = (CCheckpointData) {
    {
        { 1111, uint256("0x0000000069-244f73d70e1fd79b22fd2ed610bd6fa2e92559f542fdh26e7c1d")},
        { 33333, uint256("0x00000002dd5588a74784caa7ab0507a19ad16a236c7b10c69f00d7ddfb5d0a6")},
        { 74000, uint256("0x0000000000573993a3c9e110e34171c079d0f5f53a0e824e81e7f953b8661a20")},
        { 115000, uint256("0x0000000000291ce28027fae320c8d2b054b2e0fe44a773f3ee7b151d6bdc97")},
        { 134444, uint256("0x00000000000005b12f7d4cd315c34f7d4a594f430ac8f14c91184a0d42d2b0fe")},
        { 160000, uint256("0x00000000000099e01ea72015e75602f216fe6cb30d7f99acb05b75c0303b763")},
        { 193000, uint256("0x00000000000059f452abf7340dc6682a97f7387c17010f7f6c6c3bd83ca8a1317")},
        { 210000, uint256("0x000000000000048b95317e83192f69c70366076336c639f9b7228e9ba171342e")},
        { 216116, uint256("0x0000000000000b4f4b439e81ee46494a1945cf96c14916a4e2370f11b23df4e")},
        { 225430, uint256("0x00000000000001c108304350f74090433-7fcf79af06b0e797f065b130575932")},
        { 250000, uint256("0x0000000000000387ad1f29024c06fc2200b55f8at8f35453d7bc294dt2d214")},
        { 279000, uint256("0x000000000000001ae5c72a0b0c301f67e3afca10e819efa9011e158e9bd7e10")},
        { 295000, uint256("0x0000000000000000c4d9b4-f50f0f9d686dd69db2e03af35-10f370c64632a983")},
    }
};
```

The map stores a collection of pre-set checkpoints such that the first element of each pair is the block height, and the second is the hash of that block.

Remove all checkpoints pairs to end up with the following form:

```
checkpointData = (CCheckpointData) { { {}, } };
```

Make the same modification in the mainnet, testnet and regtest classes. Alternatively, in `validation.h` you can set the following constant to false : `static const bool DEFAULT_CHECKPOINTS_ENABLED = true;`

or in `checkpoints.cpp` you can hack the `GetLastCheckpoint` function by making it return a null pointer as follows:

```
CBlockIndex* GetLastCheckpoint(const CCheckpointData& data) {
    const MapCheckpoints &checkpoints = data.mapCheckpoints;
    for (const MapCheckpoints::value_type& i :
reverse_iterate(checkpoints)) {
        const uint256 &hash = i.second;
        BlockMap::const_iterator t = mapBlockIndex.find(hash);
        if (t != mapBlockIndex.end()) {
            // return t->second;
            return null;
        }
    }
    return nullptr;
}
```

Now the checkpoints are disabled. Nonetheless, if you want to keep them, you can pre-mine 50 blocks, and put their weights and hashes in the checkpoints, and then re-enable the checkpoints. After all, they are a powerful way for blockchain developers to protect against re-mining the whole chain.

*The message "readercoin is downloading blocks" will be displayed if the client hasn't yet downloaded all the checkpoint blocks.*

# ChainTxData

The next manoeuvre is to change some parameters in the `chaintxdata` structure, which represents the blockchain state at a specific timestamp using data of a specific block. In the Bitcoin code you will find recorded the blockchain snapshot's state at January 3, 2017 where the total number had reached 446482 mined blocks:

```
chainTxData = ChainTxData{
    // Data as of block 59e9999d3fcd1158de716d84aea1579503e5b05a73618d0bf265fa639f780a111 (height: 1353397).
    1516456933, // ^ UNIX timestamp of last known number of transactions
    19831579, // ^ total number of transactions between genesis and that timestamp
           //   (Use tx=... number in the SetBestChain below.log lines)
    0.00 // ^ estimated number of transactions per second after that timestamp
}
```

As our chain doesn't have any blocks yet we can use instead our genesis block and edit the values accordingly. Thus, the first argument 1483472411 should be changed to the timestamp used in the creation of the genesis block. The second number, 184495391, should be set to 1 because there is only the genesis transaction mined at that time. Finally, change the third value to 1.

# Rewarding

At this level, we are finally going to change the "monetary" variables of our new Cryptocurrency, such as issuance rate and the total amount of ReaderCoins circulating. As you might know, the Bitcoin protocol rewards miners by giving them freshly minted coins for doing their job. The reward is 50 new coins, which decreases constantly with time.

In `src/validation.cpp` you'll find a parameter called `nSubsidy` in the `GetBlockSubsidy` function which defines the initial value of the reward. We will be less generous, we will set this value to 10 as follows:

```
CAmount nSubsidy = 10 * COIN;
```

Remember we have set the same reward value when we have generated our genesis block in the `CreateGenesisBlock` function in `chainparams.cpp` to be consistent with the defined reward per block.

# Halving

For Bitcoin the mining reward is cut in half every 210,000 blocks, which will occur approximately every 4 years. In Bitcoin the reward, initially set to 50 BTC, fell to 25 BTC in late 2012, to 12.5 BTC in 2016 and so forth until the reward tends to 0 at the 64th halving.

To change the halving rule, in `chainparams.cpp` you have to edit the following parameter:

```
consensus.nSubsidyHalvingInterval = 210000;
```

Let's reschedule the halving interval in main and test at 100,000 blocks, equivalent to 174 days.

In `src/validation.cpp`, the halving mechanism formula is defined in the `GetBlockSubsidy` function which determines the reward amount at a specific height in the blockchain :

```
CAmount GetBlockSubsidy(int nHeight, const Consensus::Params& consensusParams)
{
    int halvings = nHeight / consensusParams.nSubsidyHalvingInterval;
    // Force block reward to zero when right shift is undefined.
    if (halvings >= 64)
        return 0;

    CAmount nSubsidy = 50 * COIN;
    // Subsidy is cut in half every 210,000 blocks which will occur approximately every 4 years.
    nSubsidy >>= halvings;
    return nSubsidy;
}
```

Such that:

- `nHeight` is the number of blocks that have been found.
- `consensusParams.nSubsidyHalvingInterval` specifies the halving's interval block.
- `nSubsidy` finally gives the number of satoshi that the coinbase may create.

After another number of blocks equivalent to `nSubsidyHalvingInterval` the reward halves again. Thus, the number of bitcoins in circulation asymptotically approaches  $nSubsidy * nSubsidyHalvingInterval * 2$ .



# Total supply

Bitcoin is designed in a such manner so it has a total circulation of approximately 21 million bitcoins (20,999,999.9769 bitcoins). In the header file `amount.h` there is a total supply `MAX_MONEY` sanity check:

```
/** No amount larger than this (in satoshi) is valid.
 *
 * Note that this constant is *not* the total money supply, which in Bitcoin
 * currently happens to be less than 21,000,000 BTC for various reasons, but
 * rather a sanity check. As this sanity check is used by consensus-critical
 * validation code, the exact value of the MAX_MONEY constant is consensus
 * critical; in unusual circumstances like a(nother) overflow bug that allowed
 * for the creation of coins out of thin air modification could lead to a fork.
 */
static const CAmount MAX_MONEY = 21000000 * COIN;
inline bool MoneyRange(const CAmount& nValue) { return (nValue >= 0 && nValue <= MAX_MONEY); }
```

Such that `COIN` is equal to  $10^8$  Satoshi (the smallest Bitcoin unit). As per our design we will set `MAX_MONEY` to 20,000,000 (rounded number).

Contrary to what one might think in the actual Bitcoin code, there is actually no **total supply parameter** which defines how many bitcoins will be generated. Nevertheless, there are rules put in place that dictate how many bitcoins will be released depending on the reward and halving rate. The following Python script simulates how the total supply is deduced from the initial reward value and the halving interval. It will print out a total of 1999999.987 units:

```
COIN = 100 * 1000 * 1000
Reward = 10
Halving = 100000
nSubsidy = Reward * COIN
nHeight = 0
total = 0
while nSubsidy != 0:
```

```
nSubsidy = Reward * COIN
nSubsidy >>= nHeight / Halving
nHeight += 1
total += nSubsidy
print "total supply is", total / float(COIN)
```

---

# POW parameters: target

As you know, proof of work is the consensus mechanism used in Bitcoin to validate (mine) the blocks. It's a repetitive brute force process aiming at finding the hash which meets specific requirements. Each hash basically gives a random number between 0 and the maximum value of a 256-bit number (which is huge). If the miner's hash is below the given target (a special hash value) then he wins. If not, he increments the nonce (completely changing the hash) and tries again.

The target is a 256-bit number that represents a threshold. Actually, it represents a threshold such that the SHA-256 hash of a candidate block's header must be lower than or equal to in order to be accepted as a valid and added into the blockchain.



In the same file, Bitcoin defines also the following parameter:

[illegible]

This specifies the minimum amount of chain work that a client must have before it will consider itself synchronized. As we are running a new chain, the minimum should be zero for both testnet and mainnet:

```
consensus.nMinimumChainWork = uint256S("0x00");
```

# New block time

In Bitcoin blocks are created and transactions get confirmed every 10 minutes. Therefore, you might assume that Bitcoin is incredibly slow network and you might want to reduce this delay in Readercoin. On our side, we will produce ReaderCoin's blocks every 2.5 minutes (as in Litecoin).

Before tinkering, let's understand the mechanic behind the minting rule of "10 minutes".

In Bitcoin, there is no parameter directly defining the block time, but actually it depends on the hashing power of the network and delay. 10 minutes was a choice made by Satoshi for stability and low latency reasons. Initially he defined an initial hash target (`powLimit`) that needs 10 minutes to be calculated, then the protocol tries to keep this block time by adjusting the blockhash so it can be found within a specific retargeting period.

Therefore the first related concept to introduce is the difficulty.

# Difficulty adjustment time interval

Difficulty is a concept used to express how difficult it is to reach the current hash target in comparison to the initial hash target used to mine the genesis block. The difficulty value is not used internally in Bitcoin but it's a metric used to express a target's height change.

In Bitcoin after each 2016 blocks, each node looks at the time stamps of the past 2015 blocks and adjusts the difficulty using the following function defined in `src/consensus/params.h` :

```
int64_t DifficultyAdjustmentInterval() const {  
    return nPowTargetTimespan / nPowTargetSpacing;  
}
```

Both parameters `nPowTargetTimespan` and `nPowTargetSpacing` are defined in `chainparams.cpp` where we will set the new following values to keep block generation at 2.5 minutes:

```
consensus.nPowTargetTimespan = 24 * 60 * 60;  
consensus.nPowTargetSpacing = 2.5 * 60;
```

The `nPowTargetSpacing` parameter indicates the average time (2.5 minutes) in which it should be possible to solve the computational problem of a new transaction block, whereas the `nPowTargetTimespan` parameter, adjusts the time interval (a day) during which the difficulty of the proof-of-work problem should be recalculated and adjusted. Therefore, difficulty is adjusted every 576 blocks ( $24 * 60 * 60 / 2.5 * 60$ ).

If the coins were on average generated too quickly since the last adjustment, the difficulty is increased. If they were generated too slowly, it is decreased. We therefore have to change in the `chainparams.cpp` file the value of the `consensus.nMinerConfirmationWindow` parameter from 2016 to 576.



# Difficulty adjustment rate

The `GetNextWorkRequired` function defined in `src/pow.cpp` is responsible for redefining difficulty by defining the next hash target to keep block production on the desired interval time:

```
unsigned int CalculateNextWorkRequired(const CBlockIndex* pindexLast, int64_t nFirstBlockTime, const Consensus::Params& params)
{
    if (params.fPowNoRetargeting)
        return pindexLast->nBits;

    // Limit adjustment step
    int64_t nActualTimespan = pindexLast->GetBlockTime() - nFirstBlockTime;
    if (nActualTimespan < params.nPowTargetTimespan/4)
        nActualTimespan = params.nPowTargetTimespan/4;
    if (nActualTimespan > params.nPowTargetTimespan*4)
        nActualTimespan = params.nPowTargetTimespan*4;

    // Retarget
    const arith_uint256 bnPowLimit = UintToArith256(params.powLimit);
    arith_uint256 bnNew;
    bnNew.SetCompact(pindexLast->nBits);
    bnNew *= nActualTimespan;
    bnNew /= params.nPowTargetTimespan;

    if (bnNew > bnPowLimit)
        bnNew = bnPowLimit;

    return bnNew.GetCompact();
}
```

Let's break down this function and jump directly to the important part recalculating the new difficulty after the difficulty adjustment time interval:

```
int64_t nActualTimespan = pindexLast->GetBlockTime() - nFirstBlockTime;
```

This calculates the time between the current time and the time of 2016 blocks (576 for ReaderCoin) ago.

```
const arith_uint256 bnPowLimit = UintToArith256(params.powLimit);
```

`bnPowLimit` is the absolute maximum target corresponding to the lowest difficulty.

`bnNew` is the new target represented in compact form (current target)  
`using : bnNew.SetCompact(pindexPrev->nBits);`

When we retarget we move the target value up or down by the ratio  $(nActualTimespan/params.nPowTargetTimespan)$ .

```
bnNew *= nActualTimespan;  
bnNew /= params.nPowTargetTimespan;
```

We cap the difficulty at the `powlimit` value to avoid using a target above the minimal target hash.

```
if (bnNew > bnPowLimit) bnNew = bnPowLimit;
```

Then the function returns the result as a 32-bit 'compact' representation of the difficulty, as the nbits we used in `chainparams.cpp` previously: `return bnNew.GetCompact();`

Thus, if for some reason 576 blocks is taking more or less than 24 hours by 20%, each node will lower or increase the difficulty (hash target value) by 20% to bring block production back to the 2.5-minute block target for a few iterations.

# Time maturity

The coinbase maturity indicator indicates a time window of 100 blocks between the creating block and the spending block. For ReaderCoin we decrease this limit from 100 to 50. In the `src/consensus/consensus.h` file we edit the following line of code:

```
static const int COINBASE_MATURITY = 50;
```

The reason behind this measure is to make coinbase transactions with less than `COINBASE_MATURITY` (50) confirmations unspendable to avoid spending coins generated for orphaned blocks.

# Block size

Welcome to the most controversial issue in Bitcoin!

Block size is simply the size, in bytes, of the serialized block. Initially Bitcoin's block had a maximum size set to 1 MB. This limit was introduced initially by Satoshi to protect the network against DOS attacks, until it was recently raised after the introduction of Segwit. Segwit, or Segregated Witness, is a technique enabling the production of blocks with a size of up to 4 MB by putting their signatures (which use roughly 60 percent of transaction space) in an extra space enabling block capacity to be scaled while maintaining backward compatibility. Segwit introduced the concept of weight instead of size, enabling old nodes (which recognise only 1 MB blocks) to only see placeholders, while nodes upgraded to Segwit are still able to see the whole block and validate the signatures (4MB).

The calculation of the weight is a bit more complicated than the simple "block size = 1 MB". The miners now need to build blocks which do not violate the conditions determined by the `CheckBlock()` function defined in `validation.cpp`:

```
// Size limits
if (block.vermagic() < block.magic() * WITNESS_SCALE_FACTOR > MAX_BLOCK_WEIGHT) {
    return SerializeSize(block, SER_NETWORK, PROTOCOL_VERSION)
    + WITNESS_SCALE_FACTOR * MAX_BLOCK_WEIGHT;
    return state.Get("bad-blob-length", false, "bad-blob-length", false, "bad-blob-length");
}
```

The `GetSerializeSize` function just computes the size in bytes of a serialized block for the network ignoring witnesses. Therefore we can deduce that the weight is proportional to size by factor of

`WITNESS_SCALE_FACTOR` as follows:  $\text{maximum size} = \text{Weight} / \text{WITNESS\_SCALE\_FACTOR}$

Any block such that `block.vtx.size()` is larger than 1 megabyte will be rejected as invalid.

You might find this weight concept a useless fancy hack but it isn't as it solves the dilemma of scaling Bitcoin without breaking the Bitcoin network consensus (backward compatibility). As we are starting a new network we can directly increase block size or keep the Segwit solution with a higher block weight. In this guide we'll opt for the second approach.

Prior to Bitcoin 0.15 the limit was defined in the `src\consensus\consensus.h` as follows:

```
static const unsigned int MAX_BLOCK_BASE_SIZE = 1000000;
```

This set the maximum limit of a Bitcoin block to 1 MB excluding witness data.

After Segwit activation, the weight parameters have been defined in `consensus.h`:

```
/** The maximum allowed size for a serialized block, in bytes (only for buffer size limits) */
static const unsigned int MAX_BLOCK_SERIALIZED_SIZE = 4000000;
/** The maximum allowed weight for a block, see BIP 141 (network rule) */
static const unsigned int MAX_BLOCK_WEIGHT = 4000000;
/** The maximum allowed number of signature check operations in a block (network rule) */
static const int64_t MAX_BLOCK_SIGOPS_COST = 80000;
static const int WITNESS_SCALE_FACTOR = 4;
```

To double the weight we have to double the values of these four parameters. This theoretically allows us to mine blocks up to 8 MB, but a more realistic maximum block size will be an occasionally rare 7.7 MB (assuming near 100% Segwit transactions). By processing big blocks, we have to make a trade-off between scalability and the growth in the blockchain's size. As the blockchain is unalterable database, reaching a big size implies higher centralisation as few storage points will be able to store data.

Congratulations, you just replicated the schism of the Bitcoin community that was proposal BIP141 (Segwit). But what are BIPs?

# BIPs: Bitcoin Improvement Proposals

In the previous chapter we introduced BIPs as a way to introduce new features to the Bitcoin protocol. Each new improvement (BIP) is activated at a future block height to give Bitcoin users time to update their software. In `chainparams.cpp` you'll find some old important BIPs (BIP34/65/66) defined as the following checkpoints:

```
consensus.BIP34Height = 227931;  
consensus.BIP34Hash = uint256S("0x0000000000024b89b412a942fe0d9fea3bb44ab7bd1b19115dd6a759c1fc08b8");  
consensus.BIP65Height = 388331; // 000000000000004c2b624ed57756c598490fd1aa2e7cc71414887910611834501  
consensus.BIP66Height = 363725; // 000000000000000379eaa19dce8c9b722d46ae6a57c2211948584290931
```

For checkpoints we can use either a block height or hash.

As you are creating a new chain from scratch, you can just use the genesis block height or hash to have these BIPs activated from the start.

```
consensus.BIP34Height = 0;  
consensus.BIP34Hash =  
  
uint256S("000001a9bbae8bb141c6941838bdacdbcf474b6ed28a0b18b2120b60a68f00ee");  
consensus.BIP65Height = 0;  
consensus.BIP66Height = 0;
```

Alongside the BIP activation time, the next eleven lines describe the activation rules (re-targeting period, needed activation threshold, version bit, fork start and ending time) for deploying soft forks :

```

consensus.nRuleChangeActivationThreshold = 1916; // 95% of 2016
consensus.nMinerConfirmationWindow = 2016; // nPowTargetTimespan / nPowTargetSpacing
consensus.vDeployments[Consensus::DEPLOYMENT_TESTDUMMY].bit = 28;
consensus.vDeployments[Consensus::DEPLOYMENT_TESTDUMMY].nStartTime = 1155145601; // January 1, 2008
consensus.vDeployments[Consensus::DEPLOYMENT_TESTDUMMY].nTimeout = 2300179999; // December 31, 2008

// Deployment of BIP65, BIP112, and BIP113.
consensus.vDeployments[Consensus::DEPLOYMENT_CSV].bit = 0;
consensus.vDeployments[Consensus::DEPLOYMENT_CSV].nStartTime = 1453060800; // May 1st, 2016
consensus.vDeployments[Consensus::DEPLOYMENT_CSV].nTimeout = 1463596800; // May 1st, 2017

// Deployment of SegWit (BIP141, BIP143, and BIP147)
consensus.vDeployments[Consensus::DEPLOYMENT_SEGWIT].bit = 1;
consensus.vDeployments[Consensus::DEPLOYMENT_SEGWIT].nStartTime = 1479168000; // November 15th, 2016.
consensus.vDeployments[Consensus::DEPLOYMENT_SEGWIT].nTimeout = 1510704000; // November 15th, 2017.

```

As we have earlier edited the `consensus.nMinerConfirmationWindow = 576;` we can therefore deduce the `nRuleChangeActivationThreshold` value by multiplying the confirmation window by 95% to get :

```
nRuleChangeActivationThreshold=574;
```

For the rest of the rules you can keep the bit element as it is and set the `nStartTime` and `ntimeout` as following:

```

consensus.vDeployments[Consensus::DEPLOYMENT_CSV].nStartTime = 0;
consensus.vDeployments[Consensus::DEPLOYMENT_CSV].nTimeout = "your
genesis time stamp";

```

Just below you'll find the assumption:

```

consensus.defaultAssumeValid =
uint256S("0x00000000000000000000000003b9ce759c2a087d52abc4266f8f4ebd6d768b89defa50a"
);

```

This assumes that the signatures in the ancestors of this block are valid. We'll need to set this hash to zero :

```
consensus.defaultAssumeValid = uint256S("0x00");
```



The modifications above should be performed for mainnet and testnet as well.

All right. Now our ReaderCoin is ready to be compiled and tested.

# Compiling and testing

Tada! The moment we have been waiting for has finally arrived.

As we have done in the beginning of this chapter to build Bitcoin, we will reproduce the same steps. Thus, let's run the following commands:

```
./autogen.sh  
./configure --with-gui=qt5 --enable-debug --disable-tests  
make && sudo make install
```

Successful compilation and installation should output as follows:

if this sounds wrong, possibly the target language is not set or unrecognized.

[illegible]

The expected end result would be to build the executable programs `readercoind` and `readercoin-qt`. You can check by running:

```
ls ./src/readercoind ./src/qt/readercoin-qt
```

Awesome, start your client using `readercoin-qt`.

A graphical interface will show up asking to set the default directory, but it will show you the Bitcoin logo and icon. Admittedly it's inconvenient to keep these graphical fingerprints, so let's get rid of them. Before going further, shut down the running ReaderCoin client.

# Design and graphics (Qt wallet)

The cherry on the cake will be defining a unique icon and image for your new-born currency, differentiating it from Bitcoin and other cryptocurrencies.

Under `/src/qt/res` directory you'll find all the other graphical resource files that you need to edit including the logo and icon (the file with an `.ico` extension) to personalise your graphical interface. Let's start by changing the background image that shows up when the graphical wallet is loaded. You can create a new image with a size of 1024\*1024 pixels – let's say a big "R" in a circle with a transparent background – using Photoshop or Gimp, and save it as `readercoin.png` under the `/src/qt/res` directory. You also have to convert this `readercoin.png` to "svg" format (`readercoin.svg`) using an online converter and save the result in the `/src/qt/res/src` directory.



When Bitcoin is run in testnet mode, the GUI uses a green image. For ReaderCoin's testnet let's choose a different colour for the

background – let's say red.

The splash picture can be also converted to an `.ico` file to serve as an icon. You can use an online converter to get your icon file. For Macintosh OSX you'll need to edit the Bitcoin file with the ICNS file extension.

Once you finish editing the graphical elements, you have to build the code again (by running `make`) to apply the graphical changes. Next time you run your ReaderCoin client's GUI, you'll see your new splash screen and icon as follows:



# Redesign of the Bitcoin-Qt wallet

Alongside editing graphical resources, you can customize the wallet interfaces to define your own. Most of the dialogs in Bitcoin-Qt are created based on the Qt framework using Qt designer. The resource files corresponding to different dialogue forms are editable files with `.ui` extensions and are located under `src/qt/forms`.

If you're not keen to use Qt, you can alternatively design and write your own graphical wallet in other languages and tools such C#/WPF ,and communicate over RPC with ReaderCoin's RPC API.

# Graphical interface addition

To start modifying the UI part, download and install the Qt dedicated IDE – Qt Creator – which includes neat tools for UI design as follow:

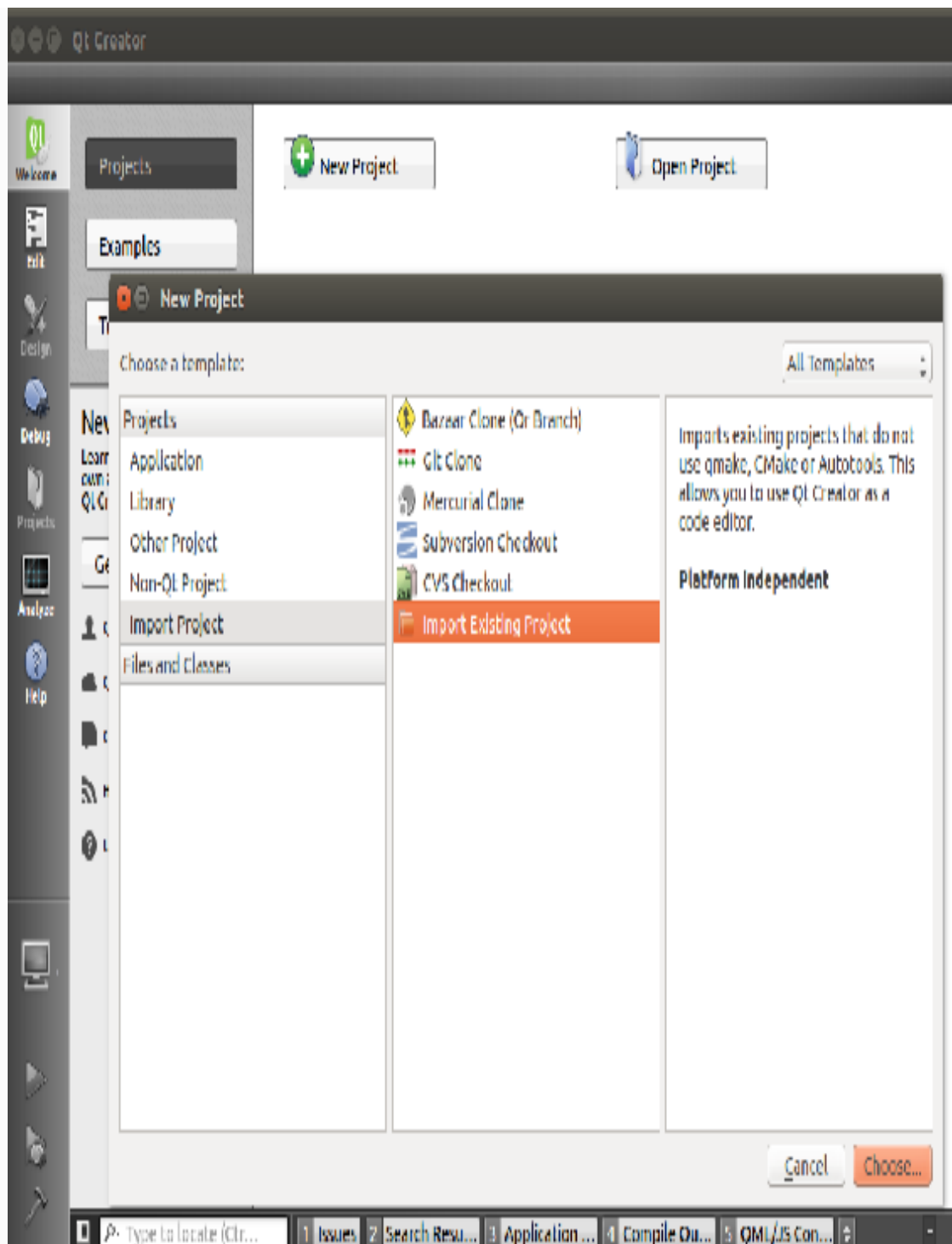
```
sudo apt-get install qtcreator
```

Let's proceed with the import of the project as follows:

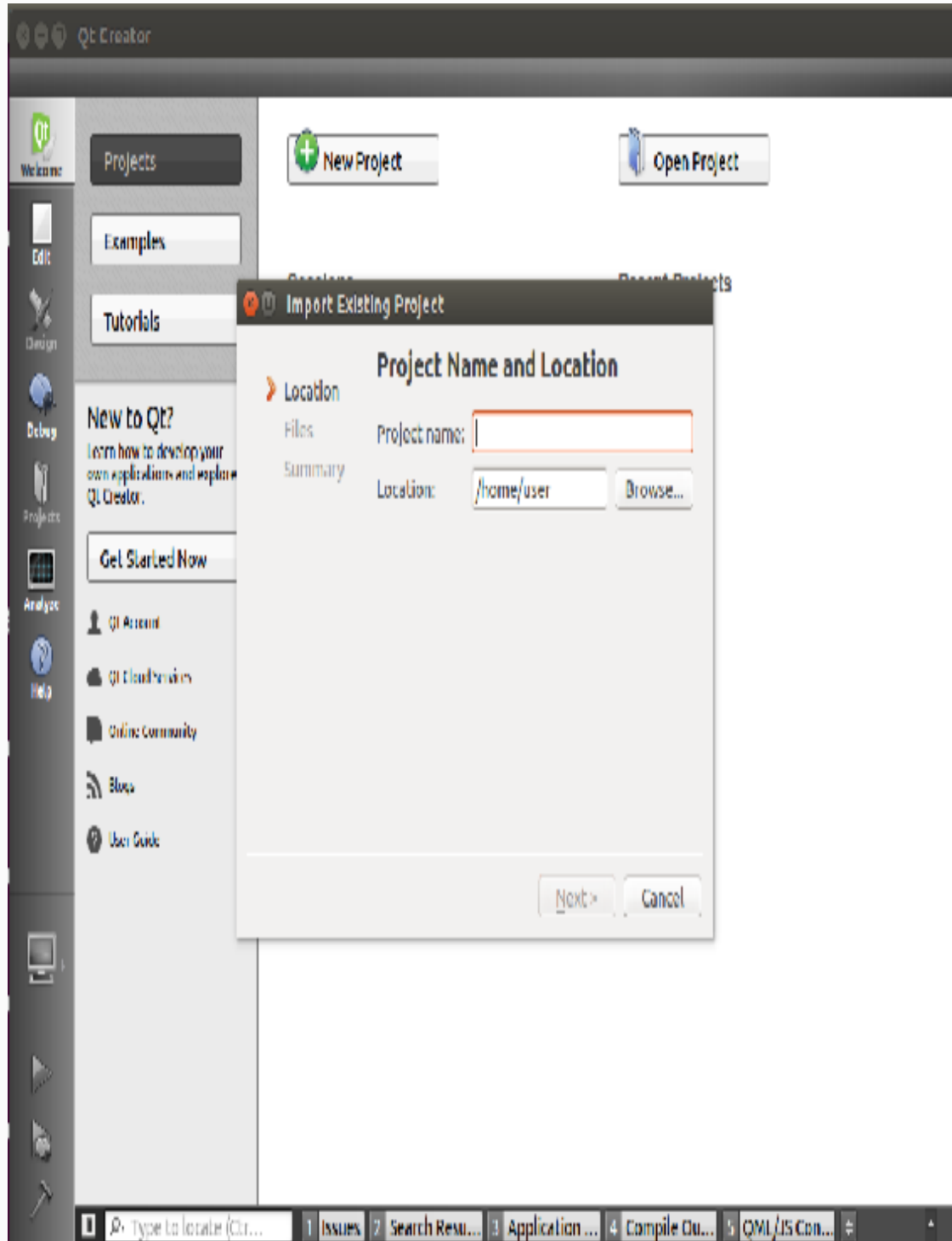


1. Run Qt Creator using: `qtcreator`

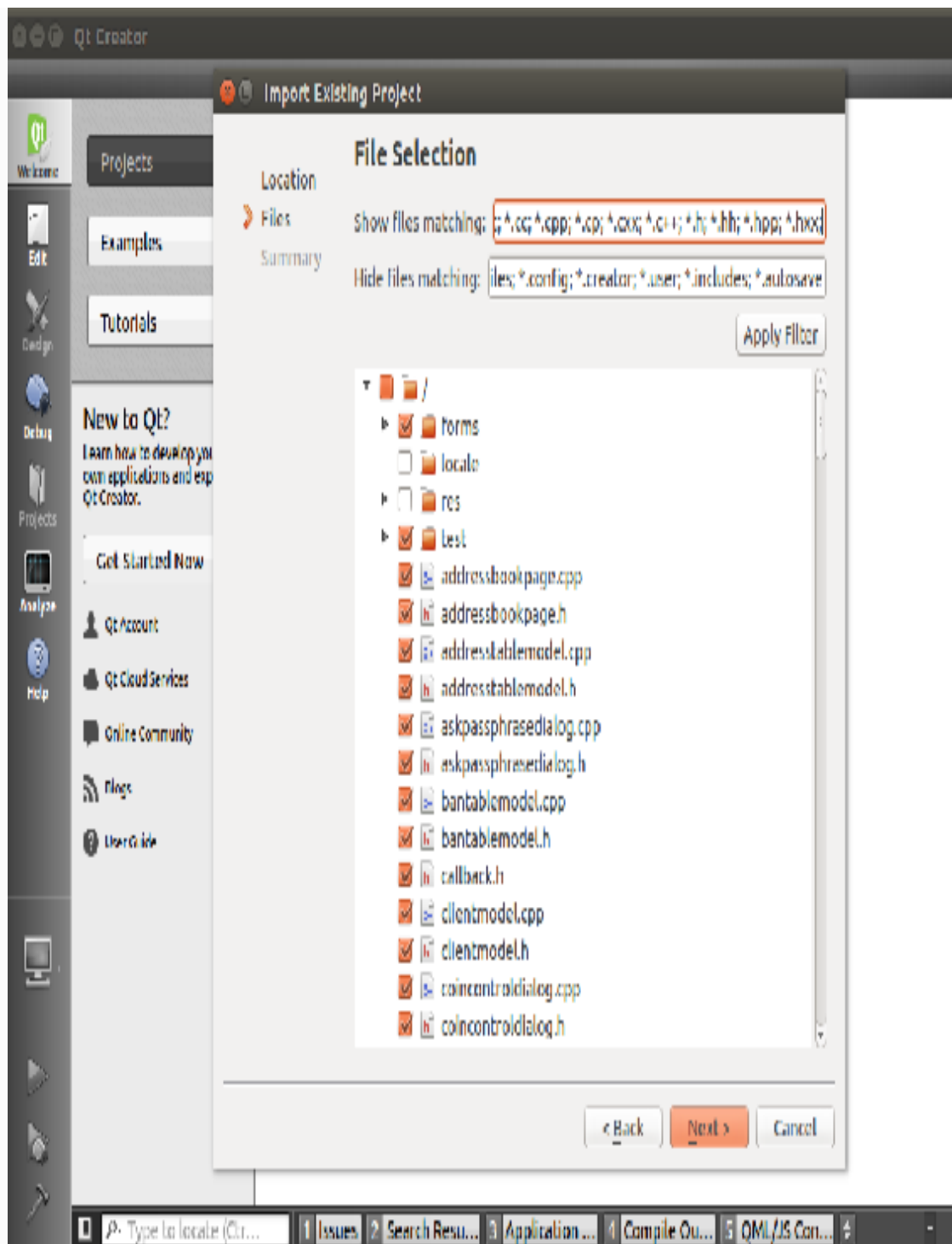
2. When the IDE shows up, in Qt Creator start a "New Project". A new dialog form will show up as shown in the figure below. Then choose "Import Project" as the template, then "Import Existing Project".



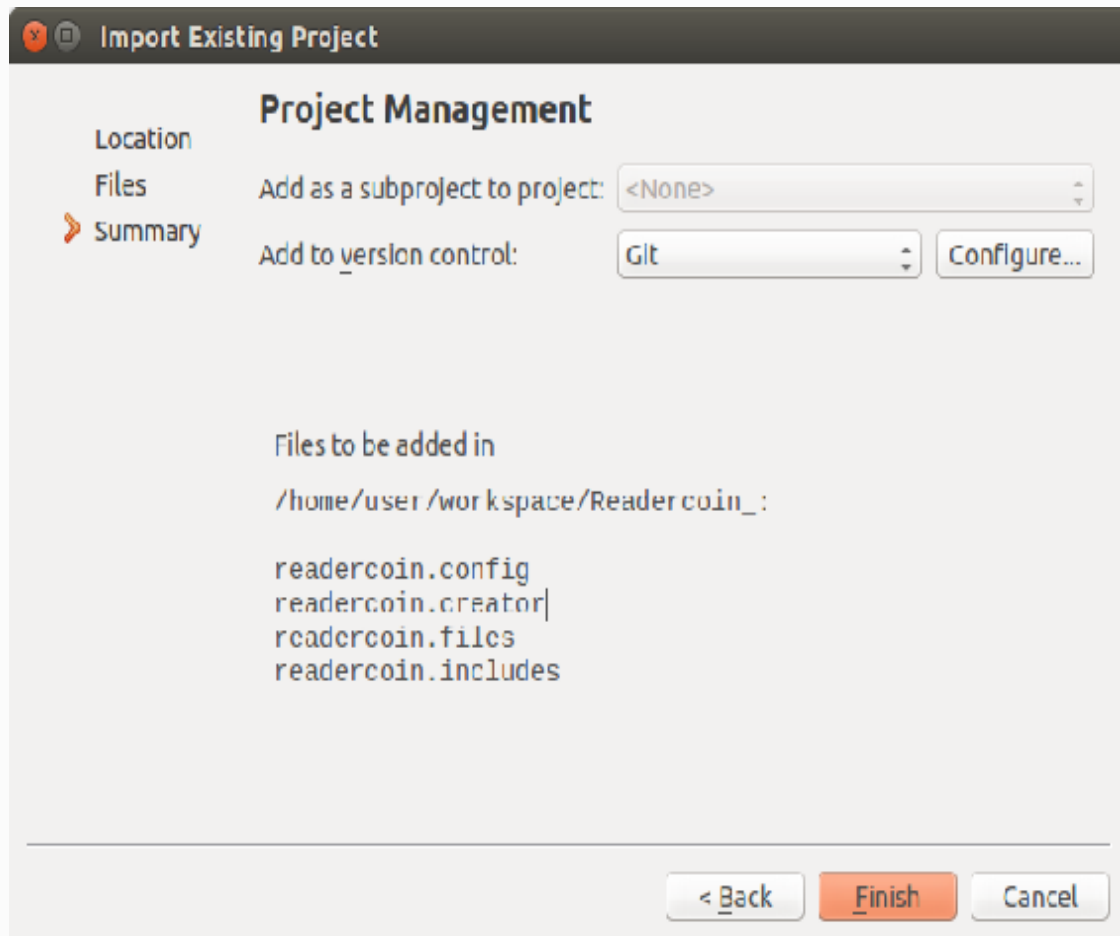
3. Enter "readercoin-qt" as project name, then select the location of src/qt in your ReaderCoin project (workspace/readercoin/src/qt).



4. A new interface will show up suggesting you choose files to import. Leave the proposed file selection as it is. We then import all the files located under the `src/` folder.

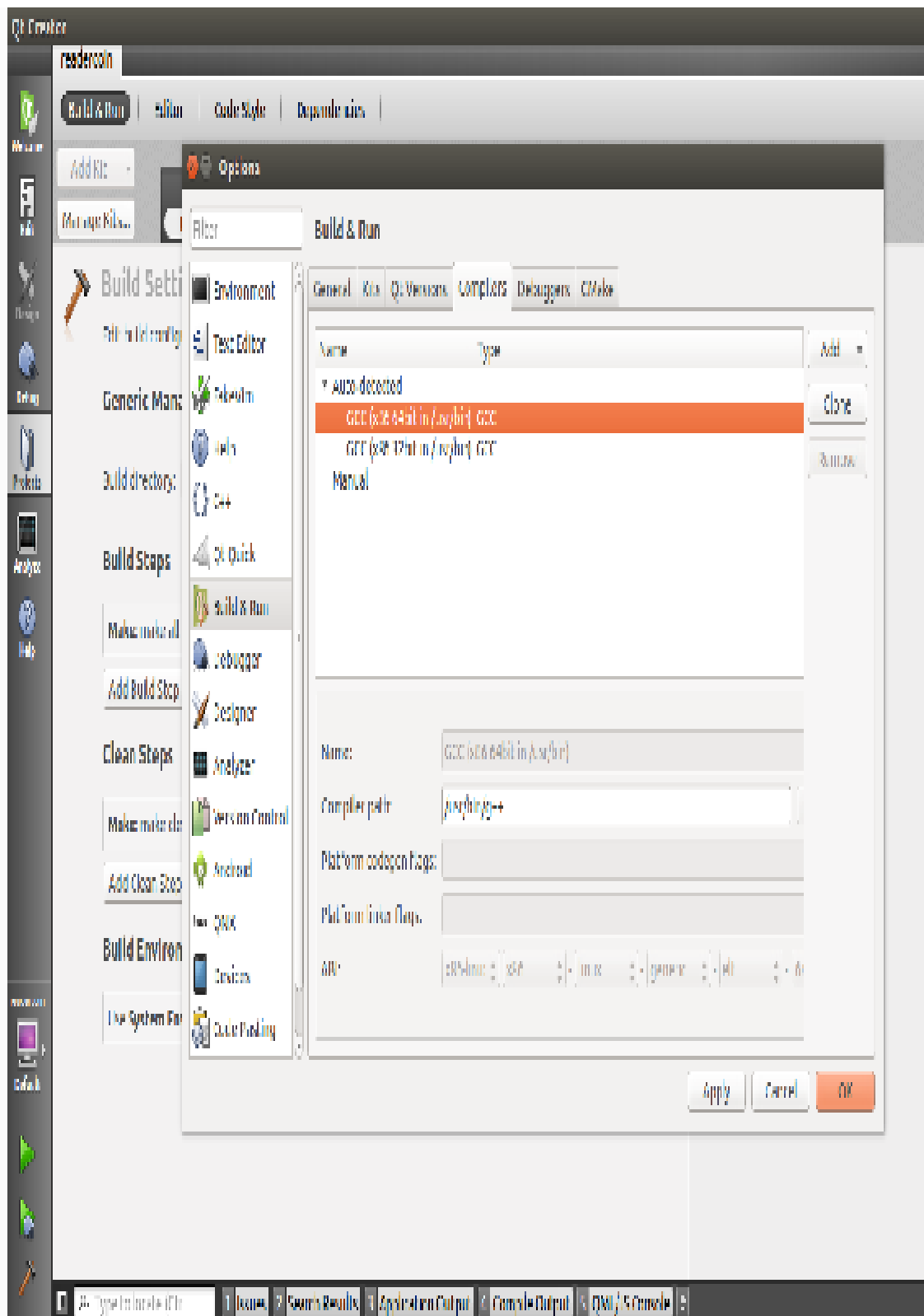


5. A "summary page" dialogue will show up where you can optionally indicate your git repository, before finally confirming the project import.



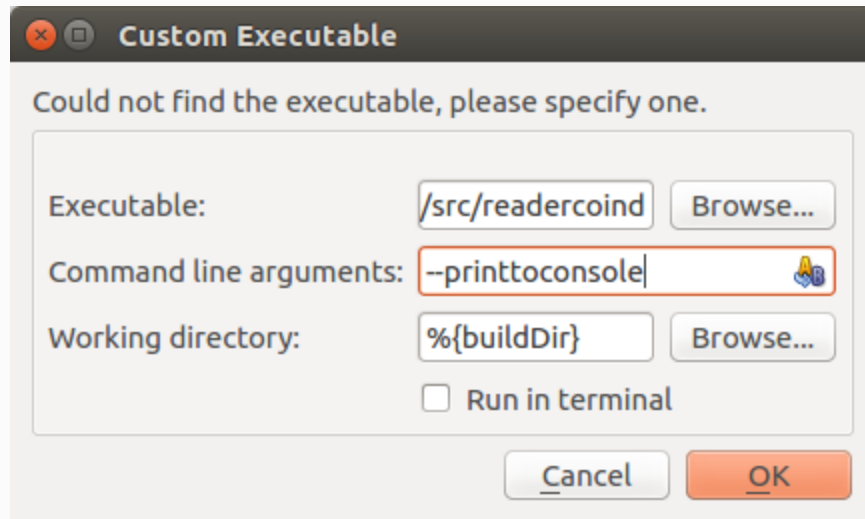
When the project is successfully imported, Qt Creator creates the project tree in the sidebar. After importing a generic project into Qt Creator, you can open it next time by selecting ".creator" file created in `src/` folder.

In the "Project" section in the left column, select "Manage Kits..", then in the section "Build & Run" select the build configuration, and set your compiler and debugger if not auto-detected.

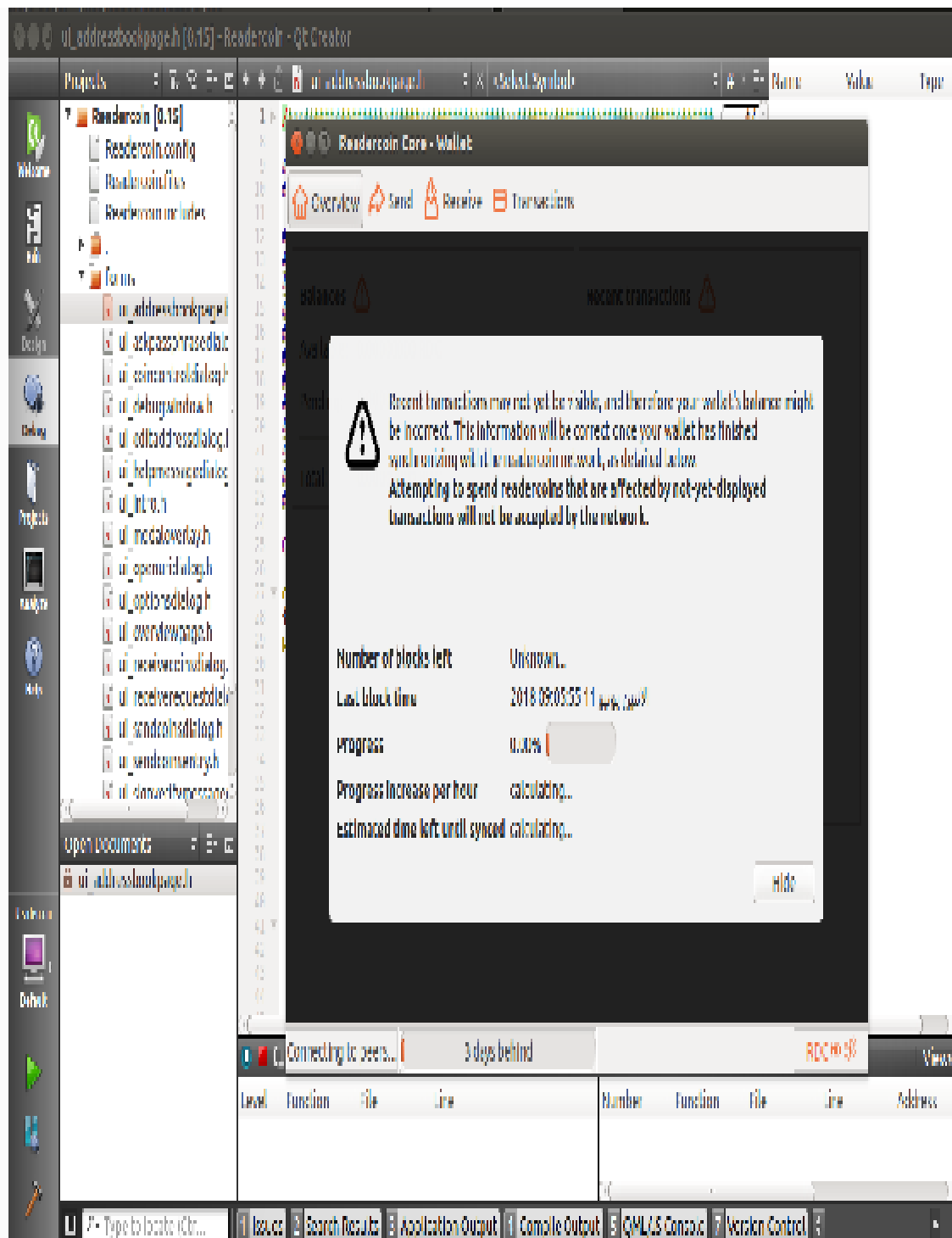




All set. You can start building the project with Qt Creator by pressing the green triangle button. It will ask you to specify the executable name and path, along with the arguments to use when running your executable. Thus we specify `src/qt/readercoin-qt` or `readercoin` as executable and `-printtconsole` as argument.



During project build you should see the log printed in the Qt Creator application output. If you edit any value using Qt Creator you'll have to recompile the code to get your modification included in your ReaderCoin client.



As you'll notice Qt Creator only gives you access to the client code. In order to edit graphical forms, start qt-designer by double clicking on the file with .ui extension located under `qt/forms/`.



compile the code by pressing the green triangle located in the bottom left of qt-creator.

# Building Readercoin network

As most of the work is done, let's put the results into practice.

At this level everything is ready to launch our ReaderCoin network. We'll need to build a network of two connected machines, called node A and node B. Before advancing further check that both machines can ping each other. Let's say nodeA has the IP address `192.168.1.3` and node B `192.168.1.8`.

At this point, we have built ReaderCoin on a single machine (node A), so we will need to build it on node B in the same manner we did in node A. For that you can clone the code source from our remote repository or just transfer the ReaderCoin folder using a USB memory stick. If successfully built, you will be ready to connect the P2P ReaderCoin nodes.

Once ReaderCoin is built successfully on both nodes, run the Qt client or `readercoin-qt -printtoconsole`.

The option `-printtoconsole` prints out the client log:

[illegible]

Running the ReaderCoin client for the first time will create a `.readercoin` directory in your home directory, along with some other

necessary files. Stop the client to create a configuration file named `readercoin.conf` in `~/.readercoin` and insert the following lines:

```
server=1
rpcuser=set a username
rpcpassword=set a password
addnode=the other node's IP
rpccallowip=192.168.0.0/16
```

Instead of using the `addnode` option in the configuration file you can run the clients on both nodes and execut the `addnode` RPC call:

```
readercoin-cli addnode 192.168.1.3:9333 onetry (in nodeA)
```

```
readercoin-cli addnode 192.168.1.8:9333 onetry (in nodeB)
```

A log entry in both peers should appear confirming the pairing similar to the following. If not, try it again or wait a bit.

```
user@ByExample-node: ~
2018-06-18 14:06:11 Bound to [::]:7333
2018-06-18 14:06:11 Bound to 0.0.0.0:7333
2018-06-18 14:06:11 init message: Loading P2P addresses...
2018-06-18 14:06:11 Loaded 0 addresses from peers.dat 0ms
2018-06-18 14:06:11 init message: Loading banlist...
2018-06-18 14:06:11 init message: Starting network threads...
2018-06-18 14:06:11 init message: Done loading
2018-06-18 14:06:11 torcontrol thread start
2018-06-18 14:06:11 net thread start
2018-06-18 14:06:11 dnsseed thread start
2018-06-18 14:06:11 Loading addresses from DNS seeds (could take a while)
2018-06-18 14:06:11 msghand thread start
2018-06-18 14:06:11 0 addresses found from DNS seeds
2018-06-18 14:06:11 dnsseed thread exit
2018-06-18 14:06:11 Imported mempool transactions from disk: 0 successes, 0 failed, 0 expired
2018-06-18 14:06:11 opencon thread start
2018-06-18 14:06:11 addcon thread start
2018-06-18 14:06:14 connect() to 192.168.1.7:9333 failed after select(): No route to host (113)
2018-06-18 14:06:14 Leaving InitialBlockDownload (latching to false)
2018-06-18 14:06:14 receive version message: /Satoshi:0.15.1/: version 70015, blocks=0, us=[::]:0, peer=0
```

You can check if both nodes are connected using : `readercoin-cli getpeerinfo`

You'll need to see all the connected peer information as shown in figure below:



```
user@ByExample-node: ~/bitcoin
user@ByExample-node:~/bitcoin$ ./src/bitcoin-cli getconnectioncount
2
user@ByExample-node:~/bitcoin$ ./src/bitcoin-cli getpeerinfo
[
  {
    "id": 23,
    "addr": "192.168.1.3:8333",
    "services": "0000000000000005",
    "relaytxes": true,
    "lastsend": 1511116445,
    "lastrecv": 1511116478,
    "bytesent": 355,
    "bytesrecv": 5636,
    "conntime": 1511116445,
    "timeoffset": 0,
    "pingtime": 0.015793,
    "minping": 0.015793,
    "version": 70012,
    "subver": "/Satoshi:0.12.1/",
    "inbound": false,
    "startingheight": 0,
    "banscore": 0,
    "synced_headers": -1,
    "synced_blocks": -1,
    "inflight": [
    ],
    "whitelisted": false
  }
]
```

It's worth noting that you can run RPC commands from the GUI without using a terminal as follow:

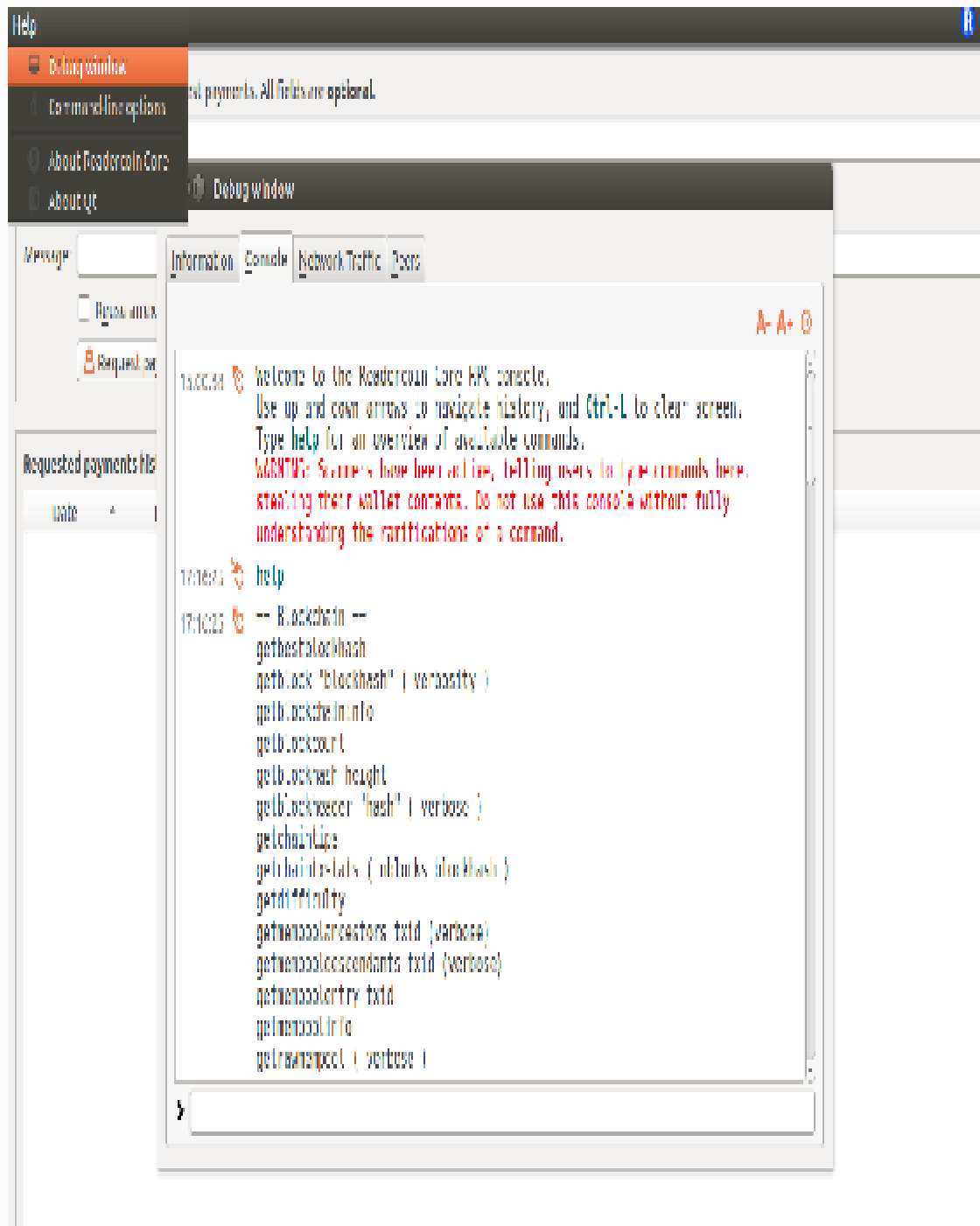
## 1. Open the ReaderCoin Wallet

2. Choose "Help"

3. Select "Debug window"

4. Select "Console"

## 5. Execute your command line without using `readercoin-cli`



Our nodes are ready to start sending and receiving ReaderCoins but as you notice we don't have any yet. Therefore, we need to start mining, either on one of the nodes or both of them.



# Readercoin mining

Bitcoin 0.15, which ReaderCoin is based on, doesn't have an internal miner, so we will need to use a dedicated mining tool to mine our blocks. Among many we will use cpuminer (minerD) which is pretty simple to use and supports CPU mining for sha256 hashing.

To install it you can get the binary file from GitHub: <https://github.com/pooler/cpuminer>

Prior to running cpuminer we need to generate new addresses in both instances using:

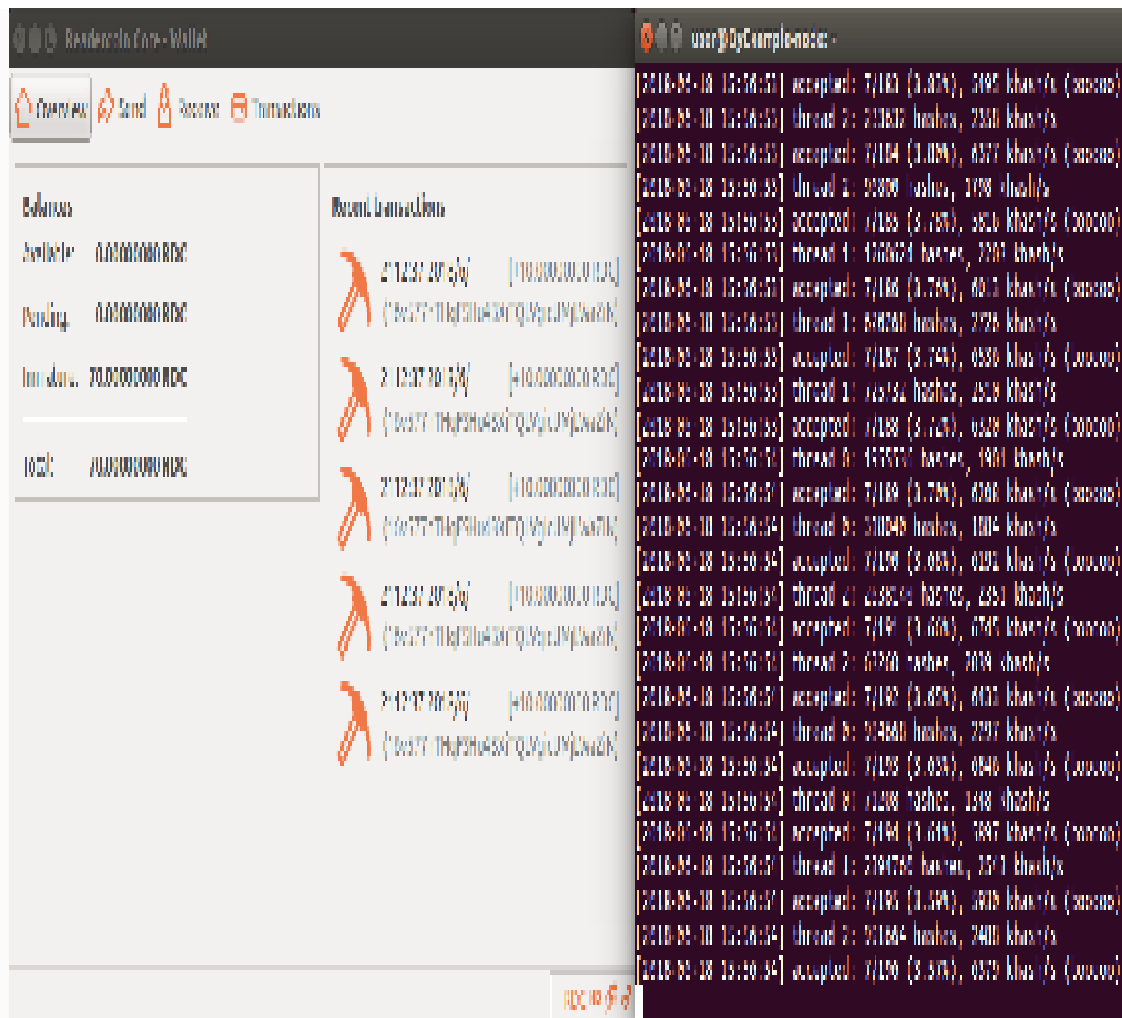
```
readercoin-cli getnewaddress
```

Then run the following command on one or both clients and provide the generated address as value for `coinbase-addr` option.

```
./minerD -o http://127.0.0.1:9332 -u user -p password -a sha256d --no-longpoll --no-getwork --no-stratum --coinbase-addr=your_address
```

Cpuminer will not start unless both nodes are connected and communicating with each other. Once working you'll be able to see a series of log entries along with notifications declaring the receipt of new transactions (genesis transactions) each with 10 ReaderCoins.





If you're facing any mining problems while using `cpuminer` you can use a more updated tool such as `cgminer` or `ccminer`. In parallel, in another terminal you can get detailed information about the progress of your blockchain using `readercoin-cli Getinfo` which returns an object containing various state information. A more specific RPC method is `getblockchaininfo` which provides you with various state information regarding blockchain processing, including the number of mined blocks, the last block hash, the chainwork and the soft fork situation.



[illegible]

As we can see above, the genesis block has a '1' difficulty and '1e00ffff0' bits. We explained earlier that Bitcoin bits represent the 'target' hash value in a special format.

Mining using a single node is called solo-mining which will be impossible to perform if the difficulty reaches a high level. In that situation it's appropriate instead to build mining pools formed by multiple mining nodes, which share the work and reward.

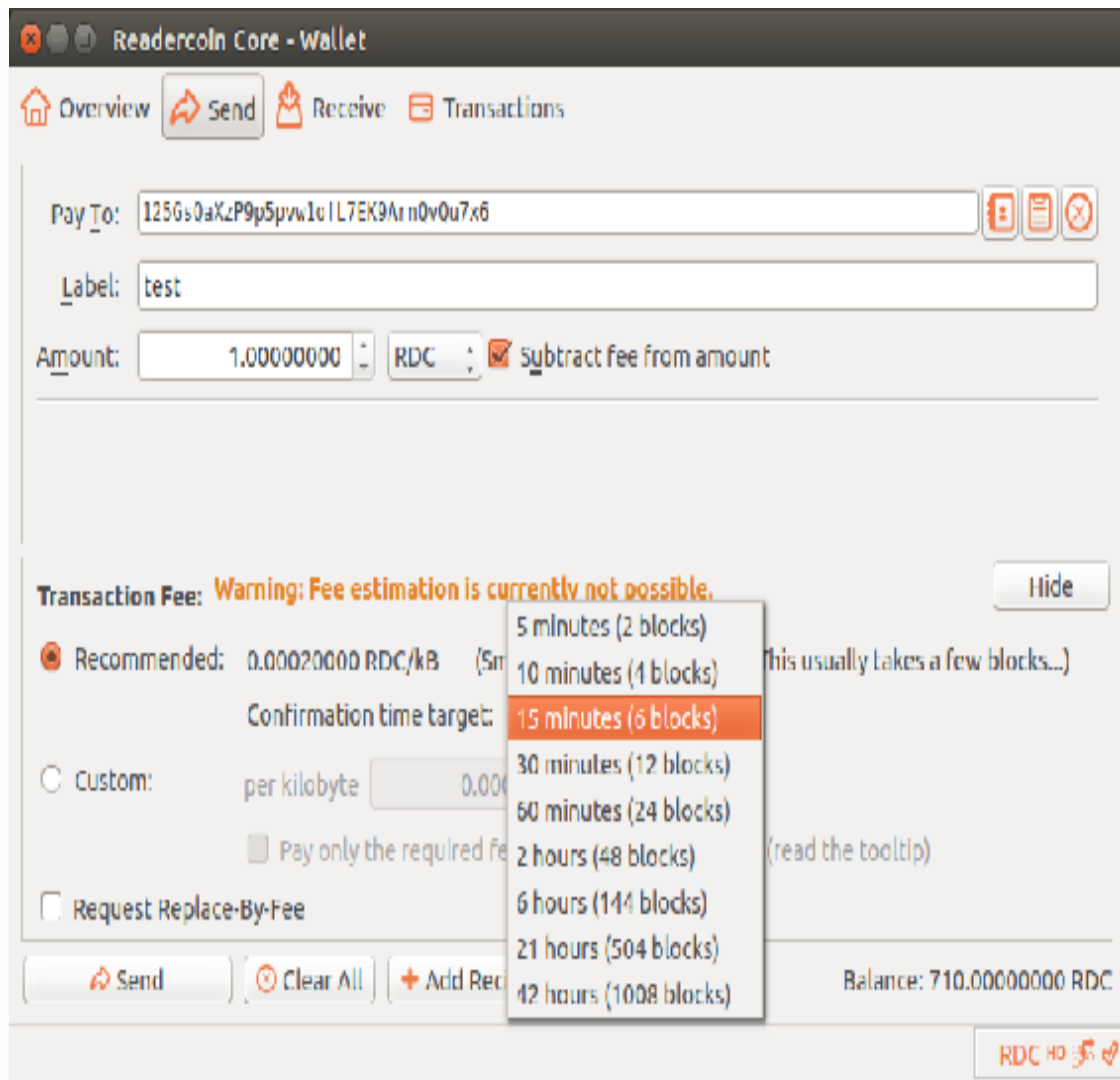
# Sending transactions

As we have set the coinbase maturity to 50 we will need to mine 50 blocks to be able to spend our mined ReaderCoins (reward set in the genesis transaction).

On node A, we can use the address generated on node B to send 1 ReaderCoin using:

```
readercoin-cli sendtoaddress <node B readercoin's address> 1
```

Or alternatively via Qt wallet as in the figure below:



You can define as for Bitcoin the transaction fee to accelerate or not your transaction's processing. You may notice the confirmation time is expressed according to our block time.

You can now enjoy extending your network and mining new coins. It's a great feeling to use your own, hand-crafted cryptocurrency to boost your understanding of blockchain, isn't it?

As a bonus we will set up an explorer to give users more visibility into the operation processing.

# Building a transaction and block explorer

Block explorers are helpful tools, providing all information about transaction processing and block details. In general, they are web applications communicating with blockchain nodes to display the network situation, avoiding you the burden of repeatedly executing RPC calls (`readercoin-cli`). There are multiple open-source Bitcoin block explorers, and by searching around GitHub you'll find projects such as `Php-Bitcoin-explorer`, `BitcoinJS explorer`, `Bitcoin-ABE`, etc. In the following guide we will opt for `Iquidus explorer`.

`Iquidus` will read the `ReaderCoin` blocks, transform and load the data into its database, and present the information via a web interface.

# Iquidus setup requirements

To set up Iquidus we need to install the following tools:

- NodeJs V6:
  - if you have newer version you can use `nvm` to manage multiple node versions: `nvm install V6`
- MonogoDB:
  - To install MongoDB run the following command line:

```
sudo apt-get install -y mongodb-org=3.6.5 mongodb-org-server=3.6.5  
mongodb-org-shell=3.6.5 mongodb-org-mongos=3.6.5 mongodb-org-  
tools=3.6.5
```

Then start MongoDB using: `sudo service mongod start`. Afterwards, follow these steps to setup the explorer's database.

# Creating Iquidus' database

Enter the MongoDB cli using the command `mongo`.

```
> use explorerdb
```

Create a user with read/write access:

```
> db.createUser( { user: "iquidus", pwd: "3xp!0reR", roles: [ "readWrite" ] }  
)
```

Note: If you're using mongo shell 2.4.x, use the following to create your user:

```
> db.addUser( { user: "username", pwd: "password", roles: [ "readWrite" ] })
```



# Installing Iquidus

Open a new terminal and clone the project's repository:

```
git clone https://github.com/iquidus/explorer.git readercoin_explorer && cd  
readercoin_explorer
```

Then install all dependencies using:

```
npm install --production
```

# Iquidus Configuration

To use the Iquidus explorer you have to edit `settings.json` with the same settings parameters set in your `readercoin.conf` file. First, we rename the settings file: `cp settings.json.template settings.json`

Then in your text editor, update the values of the following parameters:

```
"coin": "readercoin",
"symbol": "RDC",
"wallet": {
  "host": "localhost",
  "port": 9332,
  "user": "user", //Readercoin's RPC user defined in readercoin.conf
  "pass": "password"
},
```

For the parameter `"genesis_block"` you set your genesis block's hash.

For `genesis_tx` you define the hash of the genesis transaction. This value can be extracted from the `"tx"` field in the output provided by `readercoin-cli getblock your_genesis_block_hash`.

Pick any block you have in your blockchain and put its details in the following API section:

```
"api": {
  "blockindex": 0,
  "blockhash":
"000003758b8aa3b5066e2d5b2aa4f5c7516868f50d37bfd55d52ff581425adc",
  "txhash":
"6bc2585d63185acf3868cc34e0b017b3fb41c00938eb09bc52a3cf73a31ec6a8",
```

```
    "address": "1Cccex1tMVTABi9gS2VzRvPA88H4p32Vvn"  
  },  
}
```

---

Save and close your setting file.

# Syncing databases with the blockchain

To sync the explorer's local database with the blockchain information, a dedicated script `sync.js` (located in `scripts/`) is available. This script must be called from the explorer's root directory as follows:

```
node scripts/sync.js index [mode]
```

For the `mode` option you can choose one of the following modes:

- `update`: Updates index from last sync to current block.
- `check` : Checks index for (and adds) any missing transactions/addresses.
- `reindex`: Clears index then resyncs from genesis to current block.

As indicated in the official documentation, it's recommended to have this script launched via a cronjob at 2+ min intervals. For example, to update the index every minute and market data every 2 minutes use:

```
*/1 * * * * cd /path/to/explorer && /usr/bin/nodejs scripts/sync.js index  
update > /dev/null 2>&1
```

```
*/5 * * * * cd /path/to/explorer && /usr/bin/nodejs scripts/peers.js >  
/dev/null 2>&1
```



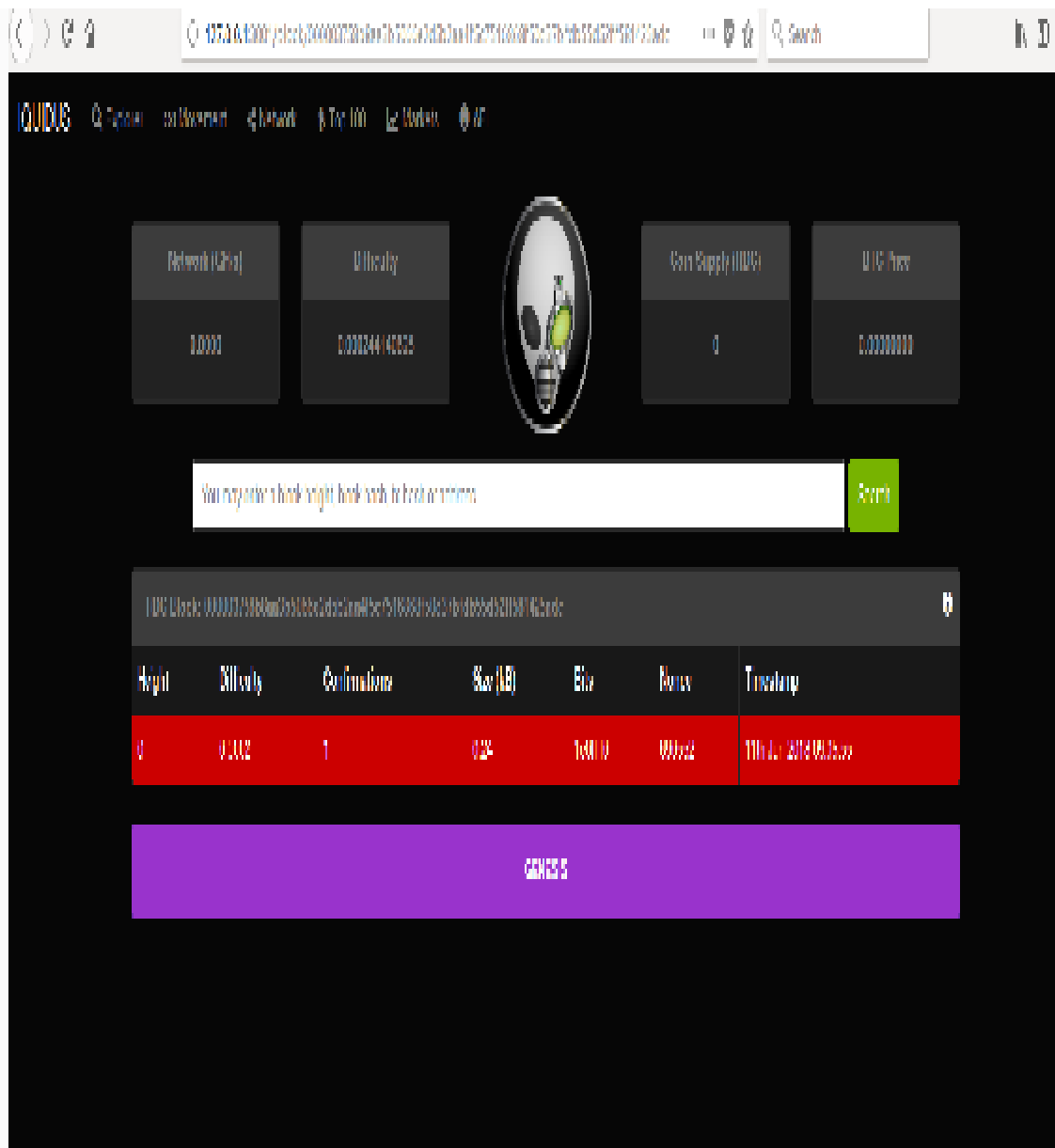
# Wallet preparation

Iquidus Explorer is ready now to connect to your node, but first we will need to run the ReaderCoin node with the option `-txindex` or set `txindex=1` in your `readercoin.conf`. This will enable getting transaction data for any transaction in the blockchain and your client will maintain an index of all transactions that have ever happened. If you have been running your client for a while but haven't had `txindex=1` set, then it might take a few hours to re-build the index.

# Starting the explorer

At this level everything is ready to start the block explorer. Open a terminal window and run within the `Iquidus` root directory `npm start` and make sure that `mongod` is already running.

If we have no errors, you can open your block explorer from `http://ip.address.of.server:3331` to access the explorer web interface.



Congratulations!!

Henceforth you can get all the blockchain information from your explorer and fetch data for a specific block, transaction or address. I hope that the steps in setting up your own altcoin explorer were easy and straightforward.

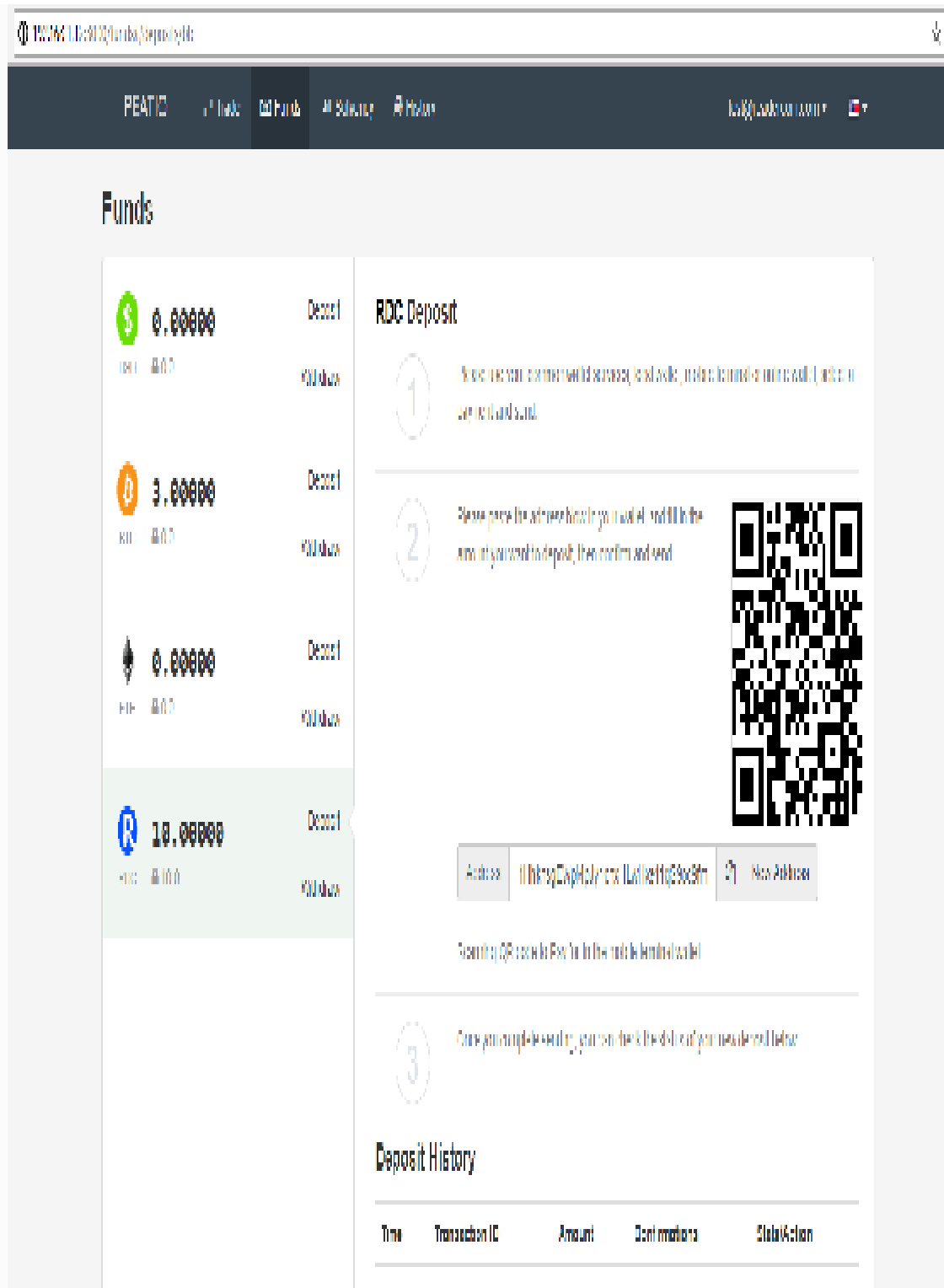


# Building an exchange market

You now have your ReaderCoin wallet, network and blockchain explorer ready to be deployed widely, all you need is to find crypto-lovers to join your project. At this level you might be thinking about trading your new altcoin to make it more attractive. To achieve that goal, you can ask well-known trading platforms to list your coin, or more excitingly, you can build your own exchange market.

To avoid you reading a long chapter, I'll just introduce you to the platform to adopt for the second option. Actually, to build your cryptocurrency exchange there is an amazing open source project called Peatio. This great platform is used in well-known trading platforms such as: Yunbi Exchange, Binance, Cex.io, etc. Peatio's code, as well as the installation procedure, are available in the official GitHub repos: <https://github.com/peatio/peatio/blob/master/doc/setup-local-ubuntu.md>.

Peatio is a complete solution, is easy to setup, and provides all the basic exchange platform operations such as cryptocurrency or fiat deposit and withdrawal, trading, KYC, and more.



The starting point before creating a new coin would be answering: what's the purpose of your new coin, do you have an original idea?

Do you solve any of Bitcoin's problems? Don't be lured into the financial temptation. You should instead focus on improving performance or implementing extended functionalities.

# Summary

The main purpose of this chapter was to present the basic steps for creating a cryptocurrency by shamelessly cloning Bitcoin. Although it only scratches the surface of building cryptocurrencies, this chapter was an opportunity to enrich your knowledge of the Bitcoin protocol and also provided a short initiation to understanding Bitcoin's source code.

This chapter concludes our short excursion into the land of Bitcoin, I hope you enjoyed it. Again, here's the GitHub repository [https://github.com/bellaj/Readercoin\\_.git](https://github.com/bellaj/Readercoin_.git) . You can catch up on any things you may have missed, or just get a better look at the code. If you face any issues you can make a pull request.

In the next chapter, we'll continue our learning journey with blockchain technology by diving deep into the very promising project – Ethereum, which is often referred to as Blockchain 2.0.

# **Chapter 4 : Peer to peer auction in Ethereum**

*Coming soon...*

# **Chapter 5 : Tontine game with Truffle and Drizzle**

*Coming soon...*

# **Chapter 6 : Blockchain-based Futures system**

*Coming soon...*

# **Chapter 7: Blockchains in business**

*Coming soon...*



# Chapter 8: Creating an ICO

*Coming soon...*

# **Chapter 9: Distributed Storage: IPFS and Swarm**

*Coming soon...*

# Chapter 10 Supply chain on Hyperledger

*Coming soon...*

# **Chapter 11 Letter of credit (LC) Hyperledger**

*Coming soon...*