UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Técnica Superior de Ingeniería Informática
Universitat Politècnica de València

Currency trading platform

Final Degree Project

Ingeniería Informática

Author: Alberto del Barrio Albelda

Director: Moisés Pastor i Gadea

September 10, 2015

Dedicated to my girlfriend for the long nights of work,
and to my family for making this possible.

Abstract

This project aims to build a web platform for intra day currency investors. Using the platform a trader is able to place orders, in real time, in the btc-e.com exchange, view the state of his wallet, look over the past orders and transactions, check the balance of his account, etc. The project fetches, stores, analyzes and transforms the information provided by the API of the exchange.
The platform enhances and adds functionality to basic operations provided by the exchange. For example it allows a trader to create several kind of orders which can expire after a defined date, be sliced between certain boundaries and much more.
A user registered in the platform can analyze the result of his orders looking at the high detailed reports automatically generated.

The design of the UI is clean and precise, with attention to the style, trying to make it as much attractive as possible for the investors.

The project consists also in designing the architecture of a high scalable web application. The system itself uses queues talking AMQP protocol for the communication between the front end and the back end components; it has long running processes, looking for changes in the market price of an asset, as well as scheduled processes, gathering information every minute.
Having all the components running properly implies a lot of work in the server side. This represents an important part of the project: decisions like which web server choose or how to ensure that a process is running all the time are widely discussed. Furthermore it explains how the application is running inside the server, which users are needed, how the MySQL database is configured, how RabbitMQ achieves persistence of the queues against crashes, etc.

But most of all, the project is about learning the elegant and simple Pythonic Way while developing a high scalable application and making my firsts steps into an trading exchange.

Keywords: cryptocurrency, trading, Bitcoin, markets, exchange, Django, Python.

Contents

Listings

List of Figures

Part I

Introduction

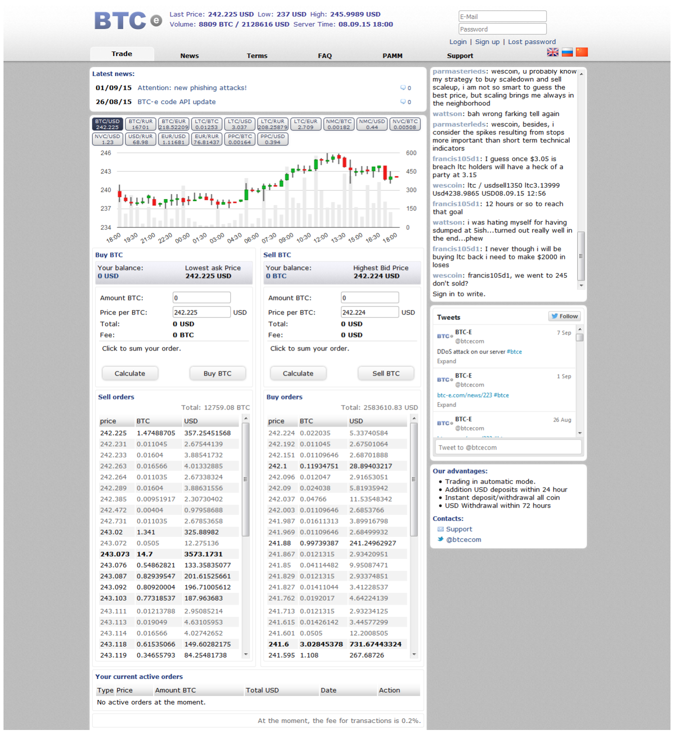
Chapter 1

Overview

The idea of making this project came while trading in btc-e.com[1].
This exchange market provides a web page where traders can change currencies. The structure of the web page is composed by five sections located in consecutive rows.
The first one, located at the top of the page, is a chart where are displayed, in form of Japanese Candels[2], the market values for an available pair of currencies. The chart covers only the lasts 24 hours of prices.
On the left side a chat is displayed, where registered users can speak.
The second section contains two boxes with two fields and two buttons each one. Here a logged user can create order to buy or sell a quantity of the selected pair of currencies.
In the third section, there are two boxes listing the nearest 40 buy and sell orders executed by the exchange.
The fourth section shows a list with the ongoing active orders of the selected pair for a logged user.
The last section, located at the bottom of the page, is one list with the lasts orders executed, both sells and buys together for the authenticated user.
This is how the page looks like.

[1]`https://btc-e.com`

[2]`http://stockcharts.com/school/doku.php?id=chart_school:chart_analysis:introduction_to_candlesticks`



BTC e Last Price: **242.225 USD** Low: **237 USD** High: **245.9989 USD**
Volume: **8809 BTC / 2128616 USD** Server Time: **08.09.15 18:00**

Latest news:

01/09/15	Attention: new phishing attacks!	0
26/08/15	BTC-e code API update	0

BTC/USD 242.225	BTC/RUR 16701	BTC/EUR 218.52209	LTC/BTC 0.01253	LTC/USD 3.037	LTC/RUR 208.25879	LTC/EUR 2.709	NMC/BTC 0.00182	NMC/USD 0.44	NVC/BTC 0.00508
NVC/USD 1.23	USD/RUR 68.98	EUR/USD 1.11681	EUR/RUR 76.81437	PPC/BTC 0.00164	PPC/USD 0.394				

Buy BTC

Your balance: **0 USD** Lowest ask Price **242.225 USD**

Amount BTC:	0	
Price per BTC:	242.225	USD
Total:	**0 USD**	
Fee:	**0 BTC**	

Click to sum your order.

Sell BTC

Your balance: **0 BTC** Highest Bid Price **242.224 USD**

Amount BTC:	0	
Price per BTC:	242.224	USD
Total:	**0 USD**	
Fee:	**0 USD**	

Click to sum your order.

Sell orders

Total: 12759.08 BTC

price	BTC	USD
242.225	**1.47488705**	**357.25451568**
242.231	0.011045	2.67544139
242.233	0.01604	3.88541732
242.263	0.016566	4.01332885
242.264	0.011035	2.67338324
242.289	0.01604	3.88631556
242.385	0.00951917	2.30730402
242.472	0.00404	0.97958688
242.731	0.011035	2.67853658
243.02	**1.341**	**325.88982**
243.072	0.0505	12.275136
243.073	**14.7**	**3573.1731**
243.076	**0.54862821**	**133.35835077**
243.087	**0.82939547**	**201.61525661**
243.092	**0.80920004**	**196.71005612**
243.103	**0.77318537**	**187.963683**
243.111	0.01213788	2.95085214
243.113	0.019049	4.63105953
243.114	0.016566	4.02742652
243.118	**0.61535066**	**149.60282175**
243.119	**0.34655793**	**84.25481738**

Buy orders

Total: 2583610.83 USD

price	BTC	USD
242.224	0.022035	5.33740584
242.192	0.011045	2.67501064
242.151	0.01109646	2.68701888
242.1	**0.11934751**	**28.89403217**
242.096	0.012047	2.91653051
242.09	0.024038	5.81935942
242.037	0.04766	11.53548342
242.003	0.01109646	2.6853766
241.987	0.01611313	3.89916798
241.969	0.01109646	2.68499932
241.88	**0.99739387**	**241.24962927**
241.867	0.0121315	2.93420951
241.85	0.04114482	9.95087471
241.829	0.0121315	2.93374851
241.827	0.01411044	3.41228537
241.762	0.0192017	4.64224139
241.713	0.0121315	2.93234125
241.615	0.01426142	3.44577299
241.601	0.0505	12.2008505
241.6	**3.02845378**	**731.67443324**
241.595	1.108	267.68726

Your current active orders

Type	Price	Amount BTC	Total USD	Date	Action
No active orders at the moment.					

At the moment, the fee for transactions is 0.2%.

parmasterleds: wescoin, u probably know my strategy to buy scaledown and sell scaleup, i am not so smart to guess the best price, but scaling brings me always in the neighborhood
wattson: bah wrong farking tell again
parmasterleds: wescoin, besides, i consider the spikes resulting from stops more important than short term technical indicators
francis105d1: I guess once $3.05 is breach ltc holders will have a heck of a party at 3.15
wescoin: ltc / usdsell1350 ltc3.13999 Usd4238.9865 USD08.09.15 12:56
francis105d1: 12 hours or so to reach that goal
wattson: i was hating myself for having sdumped at 5ish...turned out really well in the end...phew
francis105d1: I never though i will be buying ltc back i need to make $2000 in loses
wescoin: francis105d1, we went to 245 don't sold?
Sign in to write.

Tweets

BTC-E @btcecom 7 Sep
DDoS attack on our server #btce
Expand

BTC-E @btcecom 1 Sep
btc-e.com/news/223 #btce
Expand

BTC-E @btcecom 26 Aug

Our advantages:

- Trading in automatic mode.
- Addition USD deposits within 24 hour
- Instant deposit/withdrawal all coin
- USD Withdrawal within 72 hours

Contacts:

Support
@btcecom

Figure 1.1: Btc-e screenshot.

The interface is not giving enough information for a trader who wants to expend big amounts of money trading in the platform.
However the web application provides a good API 8.5.1 for the interaction with it. Thanks to this API, many web pages and services have been built around this market.
A good example of one of this service is bitcoinwisdom[3]. This web page is widely used by traders because it exposes high quality charts from many cryptocurrency markets. In these charts, a user can change the visualization period time, the kind of chart, use different indicators to try to predict the future prices of the assets and even has the possibility to put sound alarms alerting when the price reach a value.
After some months of trading with this platform I felt the necessity of other tools which help me in making more intelligent decisions based on different indicators.
After some months more, I have decided to build my own platform on top of btc-e.

[3]https://bitcoinwisdom.com

Chapter 2

Cryptocurrencies

2.1 History

2.1.1 Early Days

The first known attempt at cryptocurrencies occurred in the Netherlands, in the late 1980s, 20 years before BTC. In the middle of the night, the petrol stations in the remoter areas were being raided for cash, and the operators were unhappy putting guards at risk there. But the petrol stations had to stay open overnight so that the trucks could refuel. Someone had the bright idea of putting money onto the new-fangled smartcards that were then being developed, and so electronic cash was born. Drivers of trucks were given these cards instead of cash, and the stations were now safer from robbery.
At the same time the dominant retailer, Albert Heijn[1], was pushing the banks to invent some way to allow shoppers to pay directly from their bank accounts, which became eventually to be known as POS or point-of-sale.

2.1.2 Digital Cash

Even before this, David Chaum[2], an American cryptographer, had been investigating what it would take to create electronic cash. His views on money and privacy led him to believe that in order to do safe commerce, we would need a token money that would emulate physical coins and paper notes: specifically, the privacy feature of being able to safely pay someone hand-to-hand, and have that transaction complete safely and privately.

[1]`https://en.wikipedia.org/wiki/Albert_Heijn`
[2]`https://en.wikipedia.org/wiki/David_Chaum`

As far back as 1983, David Chaum invented the blinding formula, which is an extension of the RSA[3] algorithm which enables a person to pass a number across to another person, and that number to be modified by the receiver. When the receiver deposits her coin, as Chaum called it, into the bank, it bears the original signature of the mint, but it is not the same number as that which the mint signed. Chaum's invention allowed the coin to be modified untraceable without breaking the signature of the mint, hence the mint or bank was blind to the transaction.

All of this interest and also the Netherlands historically feverish attitude to privacy probably had a lot to do with David Chaum's decision to migrate to the Netherlands. When working in the late 1980s at CWI, a hotbed of cryptography and mathematics research in Amsterdam, he started DigiCash and proceeded to build his Internet money invention, employing amongst many others names that would later become famous: Stefan Brands, Niels Ferguson, Gary Howland, Marcel "BigMac" van der Peijl, Nick Szabo, and Bryce "Zooko" Wilcox-Ahearn.

The invention of blinded cash was extraordinary and it caused an unprecedented wave of press attention. Unfortunately, David Chaum and his company made some missteps, and fell foul of the central bank (De Nederlandsche Bank or DNB). The private compromise that they agreed to was that Digicash's e-cash product would only be sold to banks. This accommodation then led the company on a merry dance attempting to field a viable digital cash through many banks, ending up eventually in bankruptcy in 1998. The amount of attention in the press brought very exciting deals to the table, with Microsoft, Deutsche Bank and others, but David Chaum was unable to use them to get to the next level. At one point Microsoft offered Chaum $180 million to put DigiCash on every Windows PC. But Chaum that it was not enough money, and the deal fell through, and Digicash ran out of money.

2.1.3 Web Based Money

On the coattails of Digicash there were hundreds of startups per year working on this space. In the mid 1990s, the attention switched from Europe to North America for two factors: the Netscape IPO had released a huge amount of interest, and also Europe had brought in the first regulatory clampdown on

[3] `https://en.wikipedia.org/wiki/RSA_%28cryptosystem%29`

digital cash: the 1994 EU Report on Prepaid Cards, which morphed into a reaction against DigiCash.

Yet, the first great wave of cryptocurrencies spluttered and died, and was instead overtaken by a second wave of web-based monies. First Virtual was a first brief spurt of excitement, to be almost immediately replaced by PayPal[4] which did more or less the same thing. PayPal allowed the money to go from person to person, whereas First Virtual had insisted that to accept money you must "be a merchant," which was a popular restriction from banks and regulators, but people hated it. PayPal also leapt forward by proposing its system as being a hand-to-hand cash, literally: the first versions were on the Palm Pilot, which was extraordinarily popular with geeks. This geek-focus was quickly abandoned as PayPal discovered that what people really wanted was money on the web browser. Also, having found a willing user base in the eBay community, its future was more or less guaranteed as long as it avoided the bank minefield laid out for it.
So, in this moment the transfer of money was accepted as web protocol, so Chaum's ideas were more or less forgotten in the wider western marketplace, although the tradition was alive in Russia with WebMoney, and there were isolated pockets of interest in the crypto communities. In contrast, several ventures started up chasing a variant of PayPal's web-hybrid: gold on the web. The company that succeeded initially was called e-gold[5], an American-based operation that had its corporation in Nevis in the Caribbean. e-gold was a fairly simple idea: you send in your physical gold or silver, and they would credit e-gold to your account. Or you could buy new e-gold, by sending a wire to Florida, and they would buy and hold the physical gold. By tramping the streets and winning customers over, the founder managed to get the company into the black and up and growing by around 1999. As e-gold the currency issuer was offshore, it did not require US onshore approval, and this enabled it for a time to target the huge American market of 'goldbugs' and also a growing worldwide community of Internet traders who needed to do cross-border payments. With its popularity on the increase, the independent exchange market exploded into life in 2000, and its future seemed set.

2.1.4 The regulation period

e-gold however ran into trouble for its libertarian ideal of allowing anyone to have an account. While in theory this is a fine concept, the steady stream of

[4]`https://en.wikipedia.org/wiki/PayPal`
[5]`https://en.wikipedia.org/wiki/E-gold`

ponzis, HYIPs, games and other scams attracted the attention of the Feds. In 2005, e-gold's Florida offices were raided and that was the end of the currency as an effective force. The Feds also proceeded to mop up any of the competitors and exchange operations they could lay their hands on, ensuring the end of the second great wave of new monies.

In retrospect, 9/11 marked a huge shift in focus. Beforehand, the USA was fairly liberal about alternative monies, seeing them as potential business, innovation for the future. After 9/11 the view switched dramatically, although slowly; all cryptocurrencies were assumed to be hotbeds of terrorists and drugs dealers, and therefore valid targets for total control. It is probably fair to speculate that e-gold did not react so well to the shift. Meanwhile, over in Europe, they were going the other way around. It had become abundantly clear that the attempt to shut down cryptocurrencies was too successful, Internet business preferred to base itself in the USA, and there had never been any evidence of the bad things they were scared of. Successive generations of the eMoney law were enacted to open up the field, and the slightly less-high barriers remained deal killers. Which brings us forward to 2008, and the first public posting of the Bitcoin paper by Satoshi Nakamoto 2.2.3.

Bitcoin is a result of history; when decisions were made, they rebounded along time and into the design. Nakamoto may have been the mother of Bitcoin, but it is a child of many fathers: David Chaum's blinded coins and the fateful compromise with DNB, e-gold's anonymous accounts and the post-9/11 realpolitik, the cypherpunks and their libertarian ideals, the banks and their industrial control policies, these were the whole cloth out of which Nakamoto cut the invention.

2.2 Bitcoin

2.2.1 Overview

Bitcoin[6] is a digital currency also known as a cryptocurrency created by Satoshi Nakamoto. Bitcoin is like other currencies as Euro or Dollar: it can be used to purchase items locally and electronically. However, Bitcoin differs from conventional money in that it is decentralized and fully independent. No institution controls the Bitcoin Network and it is not tied to a country

[6]https://en.wikipedia.org/wiki/Bitcoin

like the US Dollar. The entire network is maintained by individuals and organizations referred to as Bitcoin Miners. Bitcoin miners process and verify its transactions through a mathematical algorithm based on the cryptographic hash algorithm SHA256.

2.2.2 History

The first version of the Bitcoin software, Version 0.1[7], was for Microsoft Windows only and had no command-line interface and it was compiled using Microsoft Visual Studio. The code was elegant in some ways and inelegant in others. The code does not appear to have been written by either a total amateur or a professional programmer; some people speculate based on this that Satoshi was an academic with a lot of theoretical knowledge but not much programming experience. Version 0.1 was remarkably complete.
Nakamoto was actively making modifications to the Bitcoin software and posting technical information on the Bitcoin Forum until his contact with other Bitcoin developers and the community gradually began to fade in mid-2010. Until a few months before leaving the Bitcoin project, almost all modifications to the source code were done by him accepting contributions relatively rarely. Then, he set up Gavin Andresen[8] as his successor by giving him access to the Bitcoin SourceForge project and a copy of the alert key.

2.2.3 Satoshi Nakamoto

Identity. Satoshi Nakamoto is the pseudonymous of a person or a group of people who designed and created the original Bitcoin software. There are no records of Nakamoto's identity or identities prior to the creation of Bitcoin. On his P2P foundation profile, Nakamoto claimed to be an individual male at the age of 37 and living in Japan, which was met with great skepticism due to his use of English and his Bitcoin software not being documented in Japanese.

Nationality The British spelling in his written work can imply that Nakamoto is British. However, he also sometimes used American spelling, which may indicate that he was intentionally trying to mask his writing style, or that he is more than one person. Investigations into the real identity of Satoshi Nakamoto have been attempted by important firms as The New Yorker, Fast Company and Newsweek.

[7] `https://github.com/bitcoin/bitcoin/releases/tag/v0.1.5`
[8] `https://en.wikipedia.org/wiki/Gavin_Andresen`

Motivation Nakamoto's work appears to be politically motivated, as quoted: "Yes, [we will not find a solution to political problems in cryptography,] but we can win a major battle in the arms race and gain a new territory of freedom for several years. Governments are good at cutting off the heads of a centrally controlled networks like Napster, but pure P2P networks like Gnutella and Tor seem to be holding their own." - Satoshi Nakamoto

2.2.4 Transactions

Each Bitcoin transaction is done between two pairs and the wallet address of both members in the transaction, it is recorded into a public log called the block chain 2.2.6. While Bitcoin can be anonymous, that doesn't mean it is. When purchasing Bitcoins on a Bitcoin trading platform or exchange, it has the user information, the Bitcoins bought can be tied back to the user.
When a user sends Bitcoins to a Bitcoin address, he can not reverse the transaction. Unlike credit cards where transaction can be disputed or reversed, Bitcoins are nonrefundable. Bitcoin can not be replaced either. If a user wallet is stored on his hard drive, he could lose his Bitcoins in many ways: being hacked, getting a virus, of loosing the computer. These lost Bitcoins can never be retrieved. That is why it is so important to take regular backups and implement measures for Bitcoin wallet security.
Furthermore, merchants cannot initiate charges on you as they can and do with credit cards. Each transaction must be initiated by the wallet holder, further underlining the advantages of the Bitcoin system.

2.2.5 Security

Proponents of Bitcoin proclaim its formidable security, and with good reason. In theory, unless 51% of the system is controlled by one party, Bitcoin is virtually unhackable. For instance, if someone wants to change a transaction or double spend a Bitcoin, he would have to obtain majority control of the system and modify every miner in this majority. When there is a disagreement in the block chain, the system overrides the minority with the data agreed upon by the majority.

51% attack

However, there are concerns that different mining companies and mining pools could be able to reach 51% of the Bitcoin hashing power and perform a so called 51% attack on the Bitcoin network[9].

[9]https://en.bitcoin.it/wiki/Weaknesses

2.2.6 Block chain

The block chain is the main innovation of Bitcoin. A block chain[10] is a transaction database shared by all nodes participating in a system based on the Bitcoin protocol[11]. A full copy of a currency's block chain contains every transaction ever executed. With this information, one can find out how much value belonged to each address at any point of the history.
Every block contains a hash of the previous block. This has the effect of creating a chain of blocks from the genesis block to the current block. Each block is guaranteed to come after the previous block chronologically because the previous block's hash would otherwise not be known. Each block is also computationally impractical to modify once it has been in the chain for a while because every block after would also have to be regenerated. These properties are what make double-spending of Bitcoins very difficult.
Honest generators only build onto a block (by referencing it in blocks they create) if it is the latest block in the longest valid chain. "Length" is calculated as total combined difficulty of that chain, not number of blocks, though this distinction is only important in the context of a few potential attacks. A chain is valid if all of the blocks and transactions within it are valid, and only if it starts with the genesis block.
For any block on the chain, there is only one path to the genesis block. Coming from the genesis block, however, there can be forks. One-block forks are created from time to time when two blocks are created just a few seconds apart. When that happens, generating nodes build onto whichever one of the blocks they received first. Whichever block ends up being included in the next block becomes part of the main chain because that chain is longer. More serious forks have occurred after fixing bugs that required backward-incompatible changes.
Blocks in shorter chains (or invalid chains) are not used for anything. When the Bitcoin client switches to another, longer chain, all valid transactions of the blocks inside the shorter chain are re-added to the pool of queued transactions and will be included in another block. The reward for the blocks on the shorter chain will not be present in the longest chain, so they will be practically lost, which is why a network-enforced 100-block maturation time for generations exists.
These blocks on the shorter chains are often called "orphan" blocks. This is because the generation transactions do not have a parent block in the longest chain, so these generation transactions show up as orphan in the "listtransactions" RPC call. Several pools have misinterpreted these messages and

[10]https://en.bitcoin.it/wiki/Block_chain
[11]https://en.bitcoin.it/wiki/Protocol_documentation

started calling their blocks "orphans". In reality, these blocks have a parent block, and might even have children. Because a block can only reference one previous block, it is impossible for two forked chains to merge.

2.2.7 Wallet

Each wallet address is unique and can not be linked to anyone unless the creator of that specific Bitcoin address reveals himself.
1Lst6Ro8r5C7QrxAuoZg1LJAuQtP3W9uV2 is an example of a unique user Bitcoin address used for receiving and sending Bitcoins. To send, receive and create Bitcoin addresses a user must have a Bitcoin wallet. A Bitcoin wallet is a software that is essentially your bank account for Bitcoin. Your wallet can hold as many Bitcoins and Bitcoin addresses you like, and you can own as many wallets you want.

2.2.8 Mining

Definition

Mining is the process of adding transaction records to Bitcoin's public ledger of past transactions. This ledger of past transactions is called the block chain as it is a chain of blocks. Bitcoin nodes use the block chain to distinguish legitimate Bitcoin transactions from attempts to re-spend coins that have already been spent elsewhere.

Mining difficulty

Mining is intentionally designed to be resource-intensive and difficult so that the number of blocks found each day by miners remains steady. Individual blocks must contain a proof of work to be considered valid. This proof of work is verified by other Bitcoin nodes each time they receive a block. Bitcoin uses the hashcash proof-of-work function.
The difficulty is the measure of how difficult it is to find a new block compared to the easiest it can ever be. It is recalculated every 2016 blocks to a value such that the previous 2016 blocks would have been generated in exactly two weeks had everyone been mining at this difficulty. This will yield, on average, one block every ten minutes. Mining a block is difficult because the SHA-256 hash of a block's header must be lower than or equal to the target in order for the block to be accepted by the network. This problem can be simplified for explanation purposes: The hash of a block must start with a certain number of zeros. The probability of calculating a hash that starts

with many zeros is very low, therefore many attempts must be made. In order to generate a new hash each round, a nonce is incremented.

Purpose

The primary purpose of mining is to allow Bitcoin nodes to reach a secure, tamper-resistant consensus. Mining is also the mechanism used to introduce Bitcoin into the system: the reward for the miners is obtained using the transaction fees as well as a the subsidy of newly created coins. These serve the purpose of disseminating new coins in a decentralized manner as well as motivating people to provide security for the system.

Mining pools

As more and more miners competed for the limited supply of blocks, individuals found that they were working for months without finding a block and receiving any reward for their mining efforts. This made mining something of a gamble. To address the variance in their income miners started organizing themselves into pools so that they could share rewards more evenly. More information about mining pools can be found here: `https://en.bitcoin.it/wiki/Mining`.

2.3 Altcoins

Altcoin[12] is a term to define all cryptocurrencies except Bitcoin, the name is an abbreviation of Bitcoin alternative. Altcoins are referred to as Bitcoin alternatives because most altcoins hope to either replace or improve upon at least one Bitcoin component. Almost all of them are forks of the Bitcoin code.
There are hundreds of altcoins, and are appearing more each day. Most altcoins are a little more than Bitcoin clones, changing only minor characteristics such as the transactions speed, distribution method, or hashing algorithm. Most of these coins do not survive for very long time. However, some altcoins are experimenting with useful features that Bitcoin does not offer. For example, Darkcoin hopes to provide a platform for completely anonymous transactions, Namecoin aims to decentralize domain-name registration for making internet censorship much more difficult and Ripple serves as a protocol that users can employ to make inter-currency payments easily. Altcoins are very important for the development and enhancement of the

[12]`https://www.cryptocoinsnews.com/altcoin/`

Bitcoin. Decentralization is one of Bitcoin's most prominent goals, and altcoins further decentralize the cryptocurrency community. Moreover, altcoins allow developers to experiment with unique features. While it is true that Bitcoin can copy these features if the developers or community desire, fully-functioning altcoins are much better "cryptocurrency laboratories" than Bitcoin's testnet. Finally, Altcoins give Bitcoin healthy competition. Altcoins give cryptocurrency users alternative options and forces Bitcoin's developers to remain active and continue innovating.
The next sections are diving into the most important and innovative altcoins, trying to explain the purpose of each one.

2.3.1 Namecoin

Origin

Created in April 2011, Namecoin was the first fork of Bitcoin so it is considered the first altcoin. Although it is a currency, Namecoin's primary purpose is to decentralize domain-name registration, which makes internet censorship much more difficult protecting free-speech rights online. The main developers are proud saying "Namecoin is the counter-example to Zooko's Triangle". Namecoin has remained one of the most successful altcoins.

Namecoin and dot-bit domains

As explained in the introduction, Namecoin is a cryptocurrency which have been created with the purpose of being used as an alternative to the domain name servers, DNS[13]. To achieve this, it proposes an approach based on the block chain instead of the traditional one of having a list of domains mapping to IPs in several servers. Like this the information of the names mapping IPs resides in the Namecoin block chain being by default decentralized. The block chain of Namecoin is special because it includes fields for storing domain registrations, record additions, modifications, etc. Therefore, the Namecoin block chain provides a transactional history for the Namecoin namespace. It is possible to browse this block chain and list all the domains using several websites, for example the Namecoin explorer[14]. At date of 23rd of July 2015 there are 8975 names. The complete list can be found using a web browser in the official page[15]/ or using namecoind software (having the complete block chain downloaded) typing "namecoind name_scan" and

[13]`https://en.wikipedia.org/wiki/Domain_Name_System`
[14]`https://explorer.namecoin.info/nbn=30/fromn=0`
[15]`http://namecoin.bitcoin-contact.org/domains.php`

filtering properly the information.
The ownership of a name is based on the ownership of a wallet, which is in turn based on public key cryptography. The Namecoin network reaches consensus every few minutes as to which names have been reserved or updated. More information about specification of domain names can be found in the dot-bit webpage[16].

Advantages of dot-bit names compared with traditional TLD

There are many advantages of using this service of name mapping. In the next lines some of them are described.

Decentralization: censorship-Resistance. One of the best benefits of using Namecoin for name mapping is the decentralization of the information. DNS servers are controlled by governments and large corporations, and could abuse of their power to censor, hijack, or spy on your Internet usage. This happens on regular basis across the world, including in countries like China and United States of America.
Dot-Bit-enabled websites are immune to these problems, because the information needed for the host name resolution is stored on your own computer. Bitcoin technology ensures that every user in the world has the same block chain data on their computer, without anyone being able to illegitimately change that data.

Transparency. All the names, including details as time to expire and history of the name are publicly available when downloading the block chain, what make it open to everyone. It is easy to query them using the namecoind software.

Security. With standard DNS, a third party can compromise a DNS server and redirect the request of the users to fake websites. Dot-Bit prevents hijacking for real. How can this work? Standard HTTPS allows CA's, or "certificate authorities" (run by governments or large corporations), to vouch for the legitimacy of a website. If a single CA gets broken into by criminals, makes a mistake, or is forced by a government, they can issue fraudulent credentials that allow someone to impersonate any website. Dot-Bit's decentralized digital records does the security job that a CA would normally do, without relying on a CA; this means that no one can easily hijack Dot-

[16]`http://dot-bit.org/Namespace:Domain_names_v2.0`

Bit-enabled websites for the same reason that no one can easily steal your Bitcoins.

Privacy. With standard DNS, the owner of the DNS server and anyone listening the user requests can deduce which websites is visiting. Instead, using Dot-Bit's digital phonebook does not generate any network traffic when a user lookup a website address, because this resolution is done in the local computer using the information stored in the block chain.

Velocity. With standard DNS, when a website switches configuration, a long period is needed until the name information get propagated all long the internet. This process can cause unnecessary downtime in many circumstances. Dot-Bit's phonebook updates within 40 minutes on average with default settings. Standard DNS servers also take time to look up a website's information, which can take long time depending on the user's bandwidth and location. Since Dot-Bit keeps the phonebook on your own computer, looking up a website usually takes few milliseconds.

Disadvantages

The main disadvantage of using Dot-bit domains is the necessity of having the entire block chain in each computer. Browsing a website using dot-bit domains can be complicated for non advanced users. There are some DNS servers which are ready to resolve bit domains, but using them all the advantages mentioned in the previous section are no longer achieved.
Another big disadvantage is the lack of documentation. It is very undocumented and the few documentation is outdated or uncompleted. This makes the process of using it very frustrating.

Registering a dot-bit domain

Registering a dot-bit domain is an easy and fast process: first a user has to download and install the Namecoin software[17], after has to create an account just populating a configuration file. Once the account is set, a user has to wait for download the entire block chain, which can take around 4 hours in July 2015. Once the block chain is downloaded, the user can register its own domain and associate it to an IP.
Namecoind is the tool for interact with the Namecoin domain system; the next commands show an example of the commands issued to register a domain.

[17] `https://github.com/vinced/namecoin`

- *namecoind name_show d/<name>* Will show the associated information, if the name is registered.
- *namecoind name_new d/<name>* Will pre-register a name, until the next 12 block will be issued.
- *namecoind name_firstupdate d/<name> <rand>* This command needs the data returned by the previous commands for finishing the registering process.

When all the process is done, the user can check the result running *namecoind name_show <yourname>*. An example of this command including the result is listed below.

```
[alberto@guacamole ~]\$ namecoind name\_show d/cryptomoneymakers
{
  "name" : "d/cryptomoneymakers",
  "value" : "{"ip":"176.9.41.35"}",
  "txid" :
      "77a5ab1f4f562682ade64e21d36f36ce034702105644643a9ca1905d4b68ff80",
  "address" : "NEm9jFwPwLJVi9E8eY7ubxGCj4mk5Yk3L9",
  "expires\_in" : 35850
}
```

Pricing

For registering a dot-bit domain a user have to pay a quantity of Namecoins. This quantity is composed by two parts: the registration fee and the transaction fee. The next table summarizes these costs.

Table 2.1: Cost of registering a dot-bit domain

Command	Registration fee	Transaction fee	Summary
name_new	0.01 NMC	0.005 NMC	Pre-order a domain
name_firstupdate	0.00 NMC	0.005 NMC	The name becomes public for 36000 blocks
name_update	0.00 NMC	0.005 NMC	Renew the domain for other 36000 blocks

Browsing dot-bit domains

Browsing a dot-bit domain requires some set up because it is not supported at the moment (and probably it will not be in the future) by the main Name servers.

nm-control. nm-control is a software written in python which is still under development but it can do some basic operations. The goal of this project is to provide a tool to manage services based on namecoind like: DNS resolution, proxy, name domain and alias, servers, registration, renewal and identity management. It can be configured to bind itself to the port 53 and resolve the DNS queries made from your computer to bit domains. It requires to have namecoind installed and synchronized with the block chain. More information as well as the source code can be found in the Github page[18].

freespechme. Using freespechme is the easiest way to browse dot-bit domains. It is a plugin for Firefox licensed under GNU license which allow a user to transparently browses sites using dot-bit domains. The final user just have to install the plugin as the usual click and install way, and configure few settings in it. To do the actual translation between names and IPs, the plugin can use a file with the information, namecoind or an external proxy. More information about the plugin can be found in the official page[19]

External proxy or DNS. The last option to browse dot-bit domains is to use and external DNS service provided by a volunteer. Configuring a UNIX system to use a third party DNS server is as easy as editing /etc/resolv.conf file adding the IP of the server.
An updated list of public DNS servers for translating dot-bit domains can be found here[20].

Possibilities. In the future, new interesting features could be added like: online identity using OpenID, file signatures, voting, web of trust, escrow and notary services.

2.3.2 Litecoin

Origin

The Litecoin Project[21] was conceived and created by Charles Lee with support of members of the Bitcoin community. It was pre-announced and was launched on October 13th, 2011. Based on Bitcoin's peer-to-peer protocol, Litecoin brings a number of features viewed by its development team as improvements over Bitcoin's implementation.

[18]`https://github.com/namecoin/nmcontrol`
[19]`http://www.freespeechme.org/`
[20]`http://dot-bit.org/How_To_Browse_Bit_Domains`
[21]`https://litecoin.info/Litecoin`

Features

Scrypt. The main feature of the Litecoin is the use of scrypt[22] as its proof-of-work algorithm. This kind of algorithms creates a computational challenge to be solved by a network of computers in order to certify a block of transactions.
Scrypt was developed in 2009 by Colin Percival. In contrast with Bitcoin's SHA-256 serves to inhibit hardware scalability by requiring a significant amount of memory when performing its calculations. The use of scrypt should delay this change, and preserve the decentralization in mining that brings a decentralized currency so much of its value and resiliency.

Transaction confirmation time. The second important feature is a reduced transaction confirmation time targeted at 2.5 minutes on average. Bitcoin confirms transactions every 10 minutes on average, and for reasonable security measures is often recommended to wait one to two hours. Litecoin's faster confirmations provide end-users with faster access to their finances, especially in time-sensitive situations.

Total amount of Litecoins. The Litecoin network will produce 84 million Litecoins, or in other words, four times as many currency units as will be issued by the Bitcoin network. For this reason, Litecoin has branded itself as "silver to Bitcoin's gold". More information regarding differences between Bitcoin and Litecoin can be found here `https://litecoin.info/User:Iddo/Comparison_between_Litecoin_and_Bitcoin`.

2.3.3 Dogecoin

History

Dogecoin[23] was created by Billy Markus from Portland, Oregon, who hoped to create a fun cryptocurrency that could reach a broader demographic than Bitcoin. In addition, he wanted to distance it from the controversial history behind Bitcoin. At the same time, Jackson Palmer, a member of Adobe Systems' marketing department in Sydney, Australia, was encouraged in Twitter to make the idea a reality.
After, Palmer purchased the domain dogecoin.com and added a splash screen, which featured the coin's logo and scattered Comic Sans text. Markus saw the site linked in an IRC chat room, and started efforts to create the currency

[22] `https://en.wikipedia.org/wiki/Scrypt`
[23] `https://en.wikipedia.org/wiki/Dogecoin`

after reaching out to Palmer. Markus based Dogecoin on the existing cryptocurrency, Luckycoin, which features a randomized reward that is received for mining a block, although this behavior was later changed to a static block reward in March 2014. In turn, Luckycoin is based on Litecoin, so uses scrypt technology in its proof-of-work algorithm.

Production schedule

Compared with other cryptocurrencies, Dogecoin has a fast initial coin production schedule: there will be approximately 100 billion coins in circulation by mid 2015 with an additional 5.256 billion coins every year thereafter. As of 10 February 2015, over 98 billion Dogecoins have been mined.

Price rising

On December 19, 2013, Dogecoin jumped nearly 300 percent in value in 72 hours, rising from US$0.00026 to $0.00095, with a volume of billions of Dogecoins per day. Three days later, Dogecoin experienced its first major crash by dropping by 80% due to large mining pools seizing opportunity in exploiting the very little computing power required at the time to mine the coin.

dogewallet compromised

On December 25, 2013, the first major theft attempt of Dogecoin occurred when millions of coins were stolen during a hacking attempt on the online wallet platform Dogewallet. The hacker gained access to the platform's filesystem and modified its send/receive page to send any and all coins to a static address. This incident spiked Tweets about Dogecoin making it the most mentioned altcoin on Twitter. To help those who lost funds on Dogewallet after its breach, the Dogecoin community started an initiative named "SaveDogemas" to help donate coins to those who lost them. Approximately one month later, enough money was donated to cover all of the coins that were lost.

Current state

By January 2014, the trading volume of Dogecoin briefly surpassed that of Bitcoin and all other crypto-currencies combined. As of 25 January 2015, Dogecoin has a market capitalization of USD 13.5 million.

Charity

The Dogecoin community and foundation have encouraged fund raising for charities and other notable causes. Money has been recollected to allow the Jamaica team[24] to go to Winter Olympic games when they could not afford the costs of the travel. Also for building a river in Kenya[25].

2.3.4 Other Altcoins

There are a lot of Altcoin designed with different focus. An exhaustive list of cryptocoins can be found at crptocoincharts[26]. On June 24th, 2015 the 20 most important coins according to that web page are reported here.

Table 2.2: List of main cryptocurrencies

Symbol	Name	Mined Coins	Difficulty	Price	Volume	Marketcap
BTC	Bitcoin	14,330,550	49402000000	1.00 BTC	62,812.93 BTC	3,753,470,752.00 USD
STR	Stellar	100,076,310,226	0	0.01 mBTC	168.30 BTC	340,755,300.80 USD
GLOBE	Globe	1,000,000,000	0	0.66 mBTC	71.45 BTC	172,953,633.60 USD
LTC	Litecoin	40,376,204	41692.2	0.02 BTC	27,762.89 BTC	160,967,126.72 USD
XUSD	CoinoUSD	100,000,000	0	3.63 mBTC	0.33 BTC	95,111,009.60 USD
UNIT	GalaxyUnit	100,000,000,000	0	1.28 uBTC	0.14 BTC	33,525,760.00 USD
DOGE	DogeCoin	97,102,803,758	23847.7	0.75 uBTC	1,547.07 BTC	19,074,874.03 USD
VIRAL	Viral	1,000,000,000	0	0.07 mBTC	3.03 BTC	17,988,665.60 USD
DRK	Darkcoin	4,598,760	3172.05	0.01 BTC	74.83 BTC	13,431,467.14 USD
NXT	Nxt	1,000,000,000	0	0.05 mBTC	247.48 BTC	13,001,708.80 USD
PPC	Peercoin	21,421,191	13.053	2.20 mBTC	171.08 BTC	12,343,399.07 USD
MINT	Mintcoin	18,789,135,808	0.027	0.37 uBTC	0.17 BTC	1,820,862.60 USD
XEM	NEM	8,999,999,999	0	0.60 uBTC	17.95 BTC	1,414,368.00 USD
FTC	Feathercoin	97,440,952	189.802	0.05 mBTC	2.22 BTC	1,276,087.34 USD
VTC	VertCoin	10,413,700	283.47	0.44 mBTC	32.28 BTC	1,203,532.88 USD
CLAM	CLAMS	624,000	0	0.01 BTC	61.46 BTC	1,176,754.18 USD
QRK	Quarkcoin	248,250,110	677.182	0.02 mBTC	70.02 BTC	1,040,995.80 USD
MEC	MegaCoin	18,353,750	10.557	0.18 mBTC	192.10 BTC	871,693.33 USD
WDC	WorldCoin	53,342,916	11.3756	0.06 mBTC	89.90 BTC	809,118.03 USD
UNO	Unobtanium	189,023	221606	0.01 BTC	1.39 BTC	531,598.07 USD

[24] http://www.theguardian.com/technology/2014/jan/20/jamaican-bobsled-team-raises-dogecoin-winter-olympics

[25] http://www.coindesk.com/dogecoin-foundation-raise-50k-kenya-water-crisis/

[26] https://www.cryptocoincharts.info

Chapter 3

Markets

3.1 btc-e.com

btc-e.com[1] is a market for trading between Bitcoins and other currencies, including the U.S. dollar, Russian ruble Litecoins, Namecoins, etc. The site has Russian (mainly) and English user interface translations.
The site was first announced on July 17, 2011 with test mode trading. Live trading began on August 7, 2011. On July 31, 2012 the service reported a security incident in which halted trading and caused financial loss to the exchange. The exchange says it covered losses from reserves and trading resumed in a matter of hours. On August 2, 2012 the service added an API for trading, but still lacks an API for Bitcoin withdrawals (something offered by every major exchange). On August 20, 2012 the service added BTC/RUR and USD/RUR trading markets.

3.2 BtcChina

BTC China[2], based in Shanghai, China, is the world's second largest Bitcoin exchange by volume as of October 2014. Founded in June 2011, it was the China's first Bitcoin exchange, and most of its customers are thought to be Chinese. In November 2013, the company had grown to 20 employees.
Company CEO Bobby Lee approached the two-person company in early 2013, and after investing his own money and attracting investors, oversaw the company's rapid expansion and marketshare growth by the end of the year. The Stanford computer science graduate, whose brother founded the cryptocurrency Litecoin, previously worked for Yahoo! in the United States, and for

[1] `https://btc-e.com`
[2] `https://www.btcchina.com/`

Walmart China as Vice President of Technology.
In November 2013, BTC China raised $5 million in Series A funding from investors Lightspeed China Partners and Lightspeed Venture Partners. On 18 December 2013, BTC China announced that it was temporarily suspending acceptance of Chinese yuan deposits, attributing the decision to government regulations, following a 5 December statement from the People's Bank of China (PBOC). On 30 January 2014, the exchange resumed accepting yuan deposits, after further studying the PBOC statement and other rules. While the PBOC prohibited banks from trading in Bitcoin, BTC China explained that they were accepting yuan into their corporate bank account, and transferring that money to their customer accounts, before it was traded for Bitcoins.

3.3 Bitstamp

Bitstamp[3] is a Bitcoin exchange based in the United Kingdom. It allows trading between USD currency and Bitcoin cryptocurrency. The company is headed by CEO Nejc Kodrič, a widely known member of the Bitcoin community, who co-founded the company in August 2011 with Damijan Merlak. The company initially operated in Slovenia, but moved its registration to the UK in April 2013.
The company was founded as a European-focused alternative to then-dominant Bitcoin exchange Mt. Gox. While the company trades in US dollars, it allows money to be deposited through the European Union's Single Euro Payments Area, allowing a relatively quick, low cost way of transferring money from European bank accounts to purchase Bitcoins.
When incorporating in the United Kingdom, the company approached the UK's Financial Conduct Authority for guidance, but was told that Bitcoin was not classed as a currency, so the exchange was not subject to regulation. Bitstamp says that it instead regulates itself, following a set of best practices to authenticate customers and deter money laundering. In September 2013, the company began requiring account holders to verify their identity with copies of their passports and official records of their home address.
Bitstamp offers an API to allow clients to use custom software to access and control their accounts. It also acts as a gateway for the Ripple payment protocol.

[3]`https://www.bitstamp.net/`

3.3.1 Service disruptions

In February 2014, the company suspended withdrawals for several days in the face of a distributed denial-of-service. Bitcoin Magazine reported that people behind the attack sent a ransom demand of 75 Bitcoins to Kodrič, who refused due to a company policy against negotiating with “terrorists”. Days after restoring service, Bitstamp temporarily suspended withdrawals for some users as a security precaution due to increased phishing attempts. European Bitcoin exchange Bitstamp suspended trading Monday after one of its active, operational Bitcoin storage wallets was "compromised" over the weekend. In a statement on its site, Bitstamp warned users not to deposit any Bitcoin to previously issued addresses.
The popular Bitcoin trading site, said to be the world’s third busiest Bitcoin exchange amounting for 6 percent of all Bitcoin transactions, said that a "small fraction" of customer Bitcoins are maintained in online systems, adding that any compromised Bitcoins can be recovered from its "cold" off line storage reserve. Co-founder and chief executive Nejc Kodric said in a tweet that the bulk of Bitstamp’s Bitcoin reserves are in cold storage, and are "completely safe". The site continued in its statement that it will "return to service". Late on Monday, Bitstamp confirmed in an emailed statement to ZDNet that "less than 19,000 Bitcoins" were stolen from the company’s operational wallet. Kodric said the Bitcoins held with Bitstamp prior to the temporary suspension of the company’s service are "completely safe and will be honored in full".
The market value of 19,000 Bitcoins represents roughly $5 million. There has been no other comment as of yet from Bitstamp or Kodric.
Many took to news-sharing and social media sites to express concern about the handling of the situation, a little over a year after the largest Bitcoin exchange Mt. Gox folded, following its claims that hackers had stolen millions of dollars worth of Bitcoins.
What happened to Bitstamp remains a mystery. No hacker group is known to have claimed responsibility for compromising the exchange’s servers. Jackson Palmer, an Adobe engineer who in his spare time created offshoot virtual currency Dogecoin, said in an email that only fraction of Bitstamp’s funds are likely to have been stolen, but that could still be a significant amount. "If someone hacks a server that’s got a hot wallet running on it, they can easily transfer out whatever balance of Bitcoin is being stored there, instantly," Palmer explained. "Most Bitcoin companies aim to store as large a percentage as possible of their Bitcoin in cold storage so that it can’t be stolen if someone malicious gains access to their server."
Bitstamp’s most recent proof-of-reserve in May showed it held 183,497 Bit-

coins in its cold wallet reserve or about $96.9 million at the time. While this figure is likely to have changed, it shows roughly the value of currency held at the exchange. Bitstamp's suspension of trading[4] has negatively affected Bitcoin's price. As of Monday afternoon in New York, the price of Bitcoin on Bitstamp was down 15 percent to $267 (at the time of publication).

3.4 Coinbase

Coinbase[5] is a Bitcoin wallet and exchange service headquartered in San Francisco California, founded by Brian Armstrong and Fred Ehrsam.
Coinbase facilitates exchange between Bitcoin and fiat currencies in twenty-six countries, and Bitcoin transactions and storage in 190 countries worldwide. Coinbase has raised a total of $106,000,000 in venture capital funding and supports 2.5 million users, 40,000 merchants, and 7,000 developer applications.

3.4.1 History

Coinbase was founded in June 2012 and enrolled in the summer 2012 Y Combinator[6] program. In October 2012 Coinbase launched the ability to buy and sell Bitcoin through bank transfers. In May 2013, Coinbase received a US$5 million Series A investment led by Fred Wilson from the venture capital firm Union Square Ventures. In December 2013, Coinbase received a US$25 million investment, from the venture capital firms Andreessen Horowitz, Union Square Ventures and Ribbit Capital.
In 2014 Coinbase grew to one million users, acquired the blockchain explorer service Blockr and the web bookmarking company Kippt, secured insurance covering the value of Bitcoin stored on their servers, and launched the vault system for secure Bitcoin storage. Throughout 2014 Coinbase also formed partnerships with Overstock, Dell, Expedia, Dish Network, Time Inc., and Wikipedia to power accepting Bitcoin payments. Coinbase also added Bitcoin payment processing capabilities to the traditional payment companies Stripe, Braintree, and Paypal.
In January 2015, Coinbase received a US$75 million investment, led by Draper Fisher Jurvetson, the New York Stock Exchange, USAA, and several banks, "apparently the first time any traditional financial institutions

[4]`http://zd.net/1JWILY9`
[5]`https://en.wikipedia.org/wiki/Coinbase`
[6]`https://en.wikipedia.org/wiki/%28company%29`

have taken direct stakes in a Bitcoin enterprise". Later in January Coinbase launched a U.S.-based Bitcoin exchange.

3.4.2 Products

Coinbase has three core products: an exchange for trading Bitcoin and fiat currency, a wallet for Bitcoin storage and transactions, and an API for developers and merchants to build applications and accept Bitcoin payments. Coinbase offers buy/sell trading functionality in 25 countries, while the wallet is available in 190 countries worldwide.
The Coinbase Exchange can be funded through a bank transfer or wire, and trades on the exchange have a maker/taker price model in which traders pay either a 0.25% fee (taker) or nothing (maker) to execute trades.

3.5 Mt. Gox

Mt. Gox was a Bitcoin exchange based in Tokyo, Japan[7]. It was launched in July 2010, and by 2013 was handling 70% of all Bitcoin transactions. In February 2014, the Mt. Gox company suspended trading, closed its website and exchange service, and filed for a form of bankruptcy protection from creditors called minji saisei, or civil rehabilitation, to allow courts to seek a buyer. In April 2014, the company began liquidation proceedings. It announced that around 850,000 Bitcoins belonging to customers and the company were missing and likely stolen, an amount valued at more than $450 million at the time. Although 200,000 Bitcoins have since been "found", the reasons for the disappearance—theft, fraud, mismanagement, or a combination of these—are unclear as of March 2014.

3.5.1 History

In late 2006, programmer Jed McCaleb thought of building a website for users of the Magic: The Gathering Online service to let them trade cards like stocks. In January 2007, he purchased the domain name mtgox.com, short for "Magic: The Gathering Online eXchange". Initially in beta release, sometime around late 2007, the service went live for around 3 months before McCaleb moved on to other projects, having decided it was not worth his time. He reused the domain name in 2009 to advertise his card game The Far Wilds.
In July 2010, McCaleb read about Bitcoin on Slashdot, and decided that

[7]`https://en.wikipedia.org/wiki/Mt._Gox`

the Bitcoin community needed an exchange for trading Bitcoin and regular currencies; a week later on 18 July, after writing an exchange website, he launched it while reusing the spare mtgox.com domain name.
As it began to take off in 2011, McCaleb announced on 6 March 2011 that he had sold MtGox to Mark Karpelès, citing the increasing demands of running an exchange. McCaleb said: "I created MtGox on a lark after reading about Bitcoins last summer. It has been interesting and fun to do. I am still very confident that Bitcoins have a bright future. But to really make MtGox what it has the potential to be would require more time than I have right now. So I have decided to pass the torch to someone better able to take the site to the next level".
By April 2013 the site had grown to handle 70% of the world's Bitcoin trades. With prices increasing rapidly, Mt. Gox suspended trading from 11–12 April for a "market cooldown". The value of a single Bitcoin fell to a low of $55.59 after the resumption of trading before stabilizing above $100. Around mid May 2013, Mt. Gox traded 150,000 Bitcoins per day, per Bitcoin Charts.

3.5.2 Security breach

On 19 June 2011, a security breach of the Mt. Gox Bitcoin exchange caused the nominal price of a Bitcoin to fraudulently drop to one cent on the Mt. Gox exchange, after a hacker allegedly used credentials from a Mt. Gox auditor's compromised computer illegally to transfer a large number of Bitcoins to himself. He used the exchange's software to sell them all nominally, creating a massive "ask" order at any price. Within minutes the price corrected to its correct user-traded value. Accounts with the equivalent of more than $8,750,000 were affected. In order to prove that Mt.Gox still had control of the coins, the move of 424,242 Bitcoins from "cold storage" to a Mt.Gox address was announced beforehand and executed in Block 132749.
In October 2011, about two dozen transactions appeared in the block chain (Block 150951) that sent a total of 2,609 BTC to invalid addresses. As no private key could ever be assigned to them, these Bitcoins were effectively lost. While the standard client would check for such an error and reject the transactions, nodes on the network would not, exposing a weakness in the protocol.

3.5.3 Insolvency and shutdown

Mt. Gox suspended withdrawals in US dollars on June 20, 2013. The Mizuho Bank branch in Tokyo that handled Mt. Gox transactions pressured Mt. Gox from then on to close its account. On July 4, 2013, Mt. Gox announced that it had "fully resumed" withdrawals, but as of September 5, 2013, few US dollar withdrawals had been successfully completed.
On 7 February 2014, all Bitcoin withdrawals were halted by Mt. Gox. The company said it was pausing withdrawal requests “to obtain a clear technical view of the currency processes”. The company issued a press release on February 10, 2014 stating that the issue was due to transaction malleability: “A bug in the Bitcoin software makes it possible for someone to use the Bitcoin network to alter transaction details to make it seem like a sending of Bitcoins to a Bitcoin wallet did not occur when in fact it did occur. Since the transaction appears as if it has not proceeded correctly, the Bitcoins may be resent. MtGox is working with the Bitcoin core development team and others to mitigate this issue”.
On 17 February 2014, with all Mt. Gox withdrawals still halted and competing exchanges back in full operation, the company published another press release indicating the steps they claim they are taking to address security issues. In an email interview with the Wall Street Journal, CEO Mark Karpelès refused to comment on increasing concerns among customers about the financial status of the exchange, did not give a definite date on which withdrawals would be resumed, and wrote that the exchange would impose "new daily and monthly limits" on withdrawals if and when they were resumed. A poll of 3000 Mt. Gox customers by CoinDesk indicated that 68% of customers were still awaiting funds from Mt. Gox. The median waiting time was between one to three months. 21% of poll respondents had been waiting for three months or more.
On 20 February 2014, with all withdrawals still halted, Mt. Gox issued yet another statement, giving no date for the resumption of withdrawals. A protest by two Bitcoin enthusiasts outside the building that houses the Mt. Gox headquarters in Tokyo continued. Citing "security concerns", Mt. Gox announced they had moved their offices to a different location in Shibuya. Bitcoin prices quoted by Mt. Gox dropped below 20% of the prices on other exchanges, reflecting the market’s estimate of the unlikelihood of Mt. Gox paying their customers.
On 24 February 2014, Mt. Gox suspended all trading, and hours later its website went offline, returning a blank page. An alleged leaked internal crisis management document claimed that the company was insolvent, after losing 744,408 Bitcoins in a theft which went undetected for years. Six other major

Bitcoin exchanges released a joint statement distancing themselves from Mt. Gox, shortly before Mt. Gox's website went offline.

3.6 Cryptsy

Cryptsy International[8] is an Internet startup managed by Project Investors, Inc. focusing on the exchange of cryptocurrencies, mainly altcoins. It currently services more than 200 different types of cryptocurrency.
The Cryptsy.com exchange opened on May 20th, 2013 and since then has seen rapid growth in both customer base and trade volume. It currently has over 270,000 registered users from all over the world with a volume of over 300k trades per day.
Cryptsy aims to provide a safe, simple, and efficient environment for users to trade cryptocurrencies with each other. Cryptsy will expand its service offerings for merchants who want to easily accept Bitcoin and other cryptocurrency payments as an alternative payment method for their e-commerce sales.

3.7 Markets comparison

As indexed before, there are many available exchange markets working with cryptocurrencies and every one of them has different strengths and weakness, making them more suitable for distinct kind of traders. The main points which characterizes them are:

- Available coins to trade with.
- Volume of the market.
- API provided.
- Capacity to ingress and withdraw coins.
- Trading fee.

These items change also in the market during the time, for example on year ago (07/2014) the exchange market btc-e.com was supporting 2 more formal currencies: GBP, CHN and 3 more cryptocurrencies: TRC, FTC, XPM. These 5 more currencies have been closed due to low volume exchange rates[9][10]. Also the fee for the transaction can change. For example btc-e.com

[8] `https://www.cryptsy.com/`
[9] `https://btc-e.com/news/207`
[10] `https://btc-e.com/news/219`

was offering discounts when trading large amounts of BTCs due to a new years campaign[11].
The table below summarizes the main points for the principal markets. The information have been taken from bitcoincharts.com[12] and coinmarkets[13] in July 2015.

Table 3.1: Market comparison.

Market	30 days volume (BTC)	30 days volume (USD)	Fee (%)	Trading pairs
BitFinex	887.512	242.145.735	0.2	6
OKCoin	8.097	-	0	4
BtcChina	656.042	176.136.570	0	3
Bitstamp	372.161	101.075.088	0.25	1
Btc-e	193.427	51.785.848	0.2	16
CoinBase	-	28.751.520	0.25	3
Cryptsy	-	169260	0.25	505

Note: The fees appearing are not exact, because are different depending on the user, volume traded and pair.
The next two pie charts give also a good overview of the markets[14].

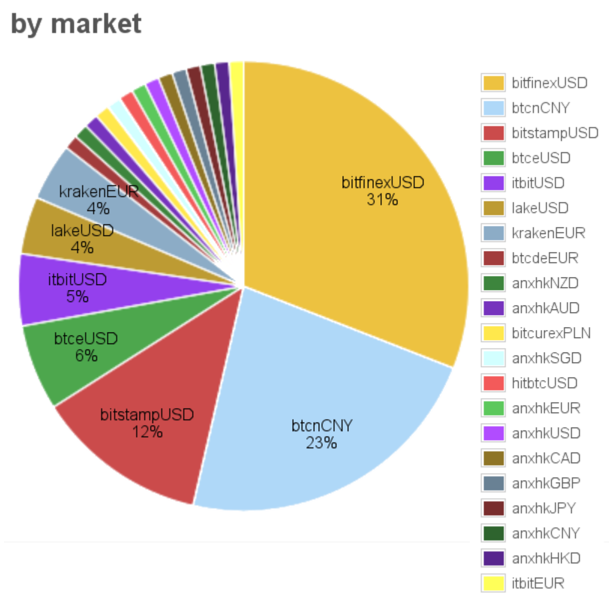


Figure 3.1: Comparison of markets by volume.

[11]https://btc-e.com/news/216
[12]http://Bitcoincharts.com/markets/
[13]http://coinmarketcap.com/currencies/Bitcoin/#markets
[14]http://Bitcoincharts.com/charts/volumepie/

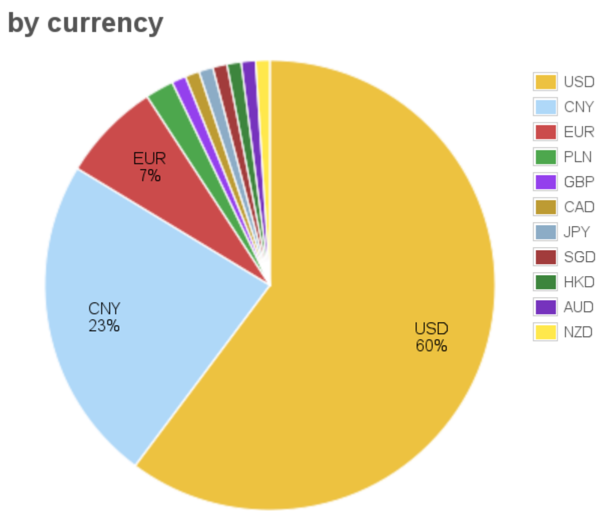


Figure 3.2: Comparison of markets by currencies.

Part II

Currency trading platform

Chapter 4

Architecture overview

4.1 Introduction

This section discuss the architecture of the project in terms of different components and the interaction between them. It also will try to explain the details behind the decision of choosing different components.
One of the key concepts that I wanted to achieve is the high scalability of the system. As this is a big project which is making high use of the resources like CPU and disk space, each decision in the design has been done thinking: what will happen if the traffic increase in 300%?
The other key concept is simplicity: I want that people contribute to the project in the future, for this I always try to write simple, clear and documented code to be able to get the attraction of possible developers.
To achieve the scalability and the simplicity I divided each single component of the project in its own block, finally obtaining: the web server, the web server gateway interface (uWSGI), the web front end, the database and the backend system. The interaction between front end and backend is done using queues, so when a user triggers a certain action in the web interface, a message is created and placed in a queue where will be picked up by some component of the backend. Doing this the system is already achieving a scalable architecture, because if the work load increases, I can add more servers consuming from a queue to assume the heavy work, and will not be a bottleneck for other parts of the system.
Other action taken to make the system scalable is the use of a management configuration tool: Chef[1]. Using Chef I have created recipes to deploy all the components of the application in a server. To illustrate how the use of Chef increase the scalability I will put an example: the servers of the project

[1]`https://www.chef.io/chef/`

are in the cloud, and we are waiting an increase of the traffic coming as a response to a marketing campaign. While looking at the graphs showing the performance of one backend component, we can see that is hitting the CPU usage limits. We will spawn a new machine, and chef it to get the packages installation, users creation, files permissions and software configuration for this machine to become part of the backend and start consuming data from the queue in minutes, just running one single command.
The next figure resumes the different components listed above.

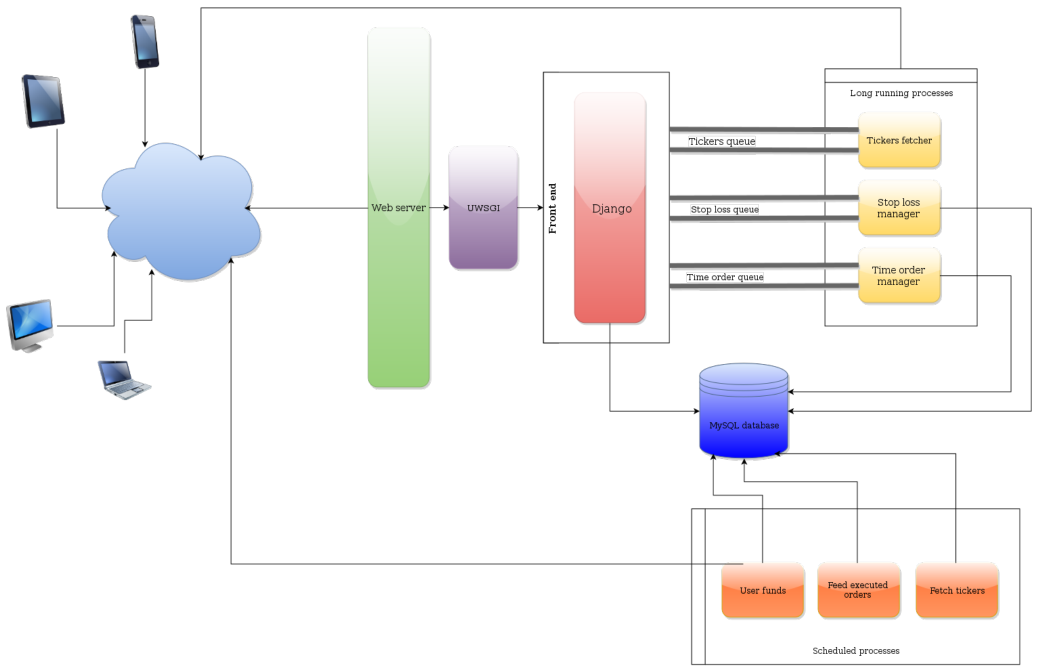


Figure 4.1: Architecture of the application.

The next sections are giving just an overview of each component, they will be highly explained in the next chapters.

4.2 Web server and gateway interface

The project uses NGINX[2] as a web server, which is nowadays widely used and is replacing Apache as a default choice. It is an open source project which also have a commercial version under subscription.
The advantages of NGINX against other traditional web servers are the focus

[2]`https://www.nginx.com/`

on concurrency, the very low use of memory and the simplicity of configuration. Thanks to its plugin architecture it can be extended and being used as a reverse proxy and load balancer.
However NGINX can not talk directly to Python applications, for this it needs a web server interface component which allows the interaction with Python as defined in the PEP 3333. Python web application frameworks have been a problem for new Python users because the choice of web framework would limit the choice of usable web servers, and vice versa. Python applications were often designed for only one of CGI, FastCGI, mod_python or some other custom API of a specific web server. The idea behind the WSGI development was to provide a low-level interface between web servers and web applications or frameworks to promote common ground for portable web application development[3]. This page explains very well why is a WSGI needed[4].
There are many WSGI servers available in the market, the most famous are: Green Unicorn[5], uWSGI[6], mod_wsgi[7] and Cherry PI[8].
In the project I choose uWSGI because it normally gives better performance than others[9], it is fully documented and easy to configure.

4.3 Front end

At the beginning of the project, I did not think in making a front end. My idea was just to do a set of scripts which were able to trade automatically in the btc-e exchange market. But step by step I had notice the lack of a place where see in a comfortable way the results of my orders and the settings of them. I also thought that if I wanted to share the project between professional traders, a graphic interface was necessary.
Considering this, I thought about develop a web using PHP and some framework to make this job easy, so I have tried Code Igniter[10] for around a week. My experience with it was not bad, but I realize that I did not like PHP as a language and I needed to invest a lot of time. For these reasons I discard this choice.

[3]https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface
[4]http://www.fullstackpython.com/wsgi-servers.html
[5]http://gunicorn.org/
[6]https://uwsgi-docs.readthedocs.org/en/latest/
[7]https://github.com/GrahamDumpleton/mod_wsgi
[8]https://github.com/cherrypy/cherrypy
[9]http://cramer.io/2013/06/27/serving-python-web-applications/
[10]http://www.codeigniter.com/

Then, I was looking into other web frameworks like Ruby on Rails[11] but in that period I did not know ruby, so the effort that I would have to invest in learning it was too much.
Finally I decided to use a Python framework. Doing a fast research the candidates could be Flask[12] and Django[13]. Obviously I choose the simplest one: Flask. It is a very lightweight and easy to use web framework and I really would recommend it for a simple web interfaces, but it was not suiting my needs because of the lack of features and community. So, at the end of the journey I choose Django, a very powerful framework with a big community around it. I am still happy with my decision, and learning how to use was real fun.
The structure of the front end and all the Django components are in-depth explained in section 6.2.

4.4 Back end

The back end is the part of the project which executes the tasks under the hood. It has two main components: programs acting as standalone processes and scheduled processes (cronjobs).
To be able to separate logically this processes from the front end, I have set up a system of queues which allows the communication between the two parts.
There are many different messages platforms like ActiveMQ, ZeroMQ, Kafka, etc. As I was a beginner in the world of message passing I opted for the most simple option RabbitMQ[14]. It was suiting my purposes because it has bindings for Python alongside great beginners tutorials, it can be configured for having data persistence and allows the definition of different kinds of queues.
For the scheduled tasks I used cronjobs, because it is a rock solid daemon simple to configure.
Each one of the components are explained with precision in 8.1.

[11]`http://rubyonrails.org/`
[12]`http://flask.pocoo.org/`
[13]`https://www.djangoproject.com/`
[14]`https://www.rabbitmq.com/`

4.5 Database

The relational data base was the most simple choice. The available choices for a free data base manager are mainly two: MySQL[15] or PostgreSQL[16]. The decision was taken in favor of MySQL because I had work with this data base before. This is one of the parts which I would like to change in the future for other kind of data base which can scale better, probably in the direction of NOSQL data bases like MongoDB[17] or key-value pairs like Redis[18].
Actually I am not using MySQL but MariaDB[19], which is a fork of MySQL developed by the community which remains with GNU license. I did not choose it, but it is the default data base server in CentOS 7, and for a basic usage it is completely equal to MySQL, so I got it as was fulfilling my purposes.

[15]`https://www.mysql.com/`
[16]`http://www.postgresql.org/`
[17]`https://www.mongodb.org/`
[18]`http://redis.io/`
[19]`https://en.wikipedia.org/wiki/MariaDB`

Chapter 5

Project deployment

5.1 Server

5.1.1 First server

When the project was started, the idea was to develop a backend system, without graphic interface. Looking at the hosting providers I chose 1and1[1] due to his low fares. The server was a VPS[2] equipped with 2 virtual Cores, 2 GB of guaranteed RAM being able to use 4 GB of RAM in some moments and 150 GB of space. This machine was reasonable at the beginning, running smoothly Django, scheduled processes and a git server.
But in the moment that RabbitMQ was installed, it was starting to have problems allocating memory, complaining that it could not fork anymore. Looking at the memory used everything seemed to be fine, but diving inside the problem I realized that I was limited in the number of total threads that I could actually run at the same time, the limit was only 128. After some discussions with customer support I decided to move to a dedicated server.

5.1.2 Current server

The server that nowadays is serving the project was rented in a Christmas offer at low price: 30 eur/month. I chose this server because of the good experience working with the provider: Hetzner[3]. This German company can provide low prices buying used servers and memory.
The machine is located in Nuremberg (Germany) and it is suited with a Intel Core i7-2600 with 16 GB of RAM memory divided in 4 blocks of 4096 MB

[1]`https://www.1and1.com/`
[2]`https://en.wikipedia.org/wiki/Virtual_private_server`
[3]`https://www.hetzner.de/`

each one and using 2 HDD with a capacity of 3 TB each one.
Because of the focus on the data reliability, the hard drives are set up using RAID 1[4]. A RAID 1 set up consists in an exact copy (or mirror) of one disk into the other one. This layout is useful when read performance or reliability is more important than the resulting data storage capacity. This set up ensures the data persistence. If one of the two HDDs breaks suddenly, it can be changed allowing the system to continue running using the healthy disk.

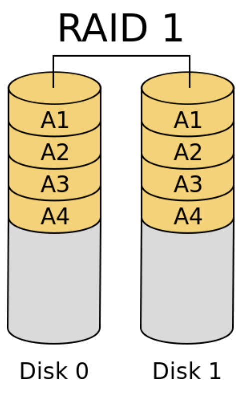


Figure 5.1: RAID 1

The system is running the last version of the Community Enterprise OS: CentOS 7, which is based on Red Hat 7. This operating system was chosen because its use is very common in the server's landscape, it has 10 years of official support and is the one which I am working everyday.

5.2 Configuration management: Chef

Configuring the server is one of the key points for a successful project. The desire of every developer is to have a server secure, reliable with good performance. Achieving this is not a trivial task due to the many different components which have to be properly configured.
Traditionally the servers were configured manually, setting up all the services, and luckily using pre-configured files for them like: sshd configuration, web server configuration, etc. Nowadays this task can be done using configuration management tools. A configuration management tool is a software that makes easier the task of configuring a server. There are many different out there: Puppet, Chef, Ansible, Salt, etc. The choose of the tool finish

[4]`https://en.wikipedia.org/wiki/Standard_RAID_levels#RAID_1`

in favor of Chef because I have already know it and I have created many cookbooks for my own purposes and for the community.
Chef[5] is a configuration management tool licensed under Apache License. It can be used for small environments (as can be the case of a personal laptop) or for large and complicate environments with hundreds of servers. It is written in Ruby and Erlang and use a pure Ruby DSL for achieve the desired state of a machine.
Chef uses cookbooks, roles and nodes as main key concepts. So, inside the cookbooks there are recipes, attributes and templates: a recipe contains a set of rules to achieve some specific configuration in a machine, and the templates are files which can receive parameters changing its content. They are normally used for rolling out configurations of services.
The recipes used in the server are:

- Installing and configuring NGINX and RabbitMQ.
- Creating the user who serves the code.
- Creating a Python virtual environment.
- Installing the cronjobs for the scheduled tasks.

The analysis of the Chef code is probably outside of the scope of the project, for this I decided to show only few snippets of the code which do precise tasks. The whole set of recipes can be found in the appendix 10.1.4. This snippet shows how the main packages are installed.

```
%w(
  python-virtualenv git python-pip gcc mariadb-devel enca librabbitmq
  rabbitmq-server
  ).each do |p|
    package p do
    action :install
  end
end
```

The next one creates a Python virtual environment where the code is living.

```
python_virtualenv node['crytomoneymakers']['venv_path'] do
  interpreter 'python2.7'
  owner      'cryptomoneymaker'
  group      'cryptomoneymaker'
  options    '--system-site-packages'
```

[5]https://www.chef.io/

```
  action     :create
end

python_pip 'django' do
  version   '1.6'
  virtualenv node['crytomoneymakers']['venv_path']
  action    :install
end

%w(uwsgi mysql-python pycrypto Pillow iconv django-datetime-widget
    pika).each do |p|
  python_pip p do
    virtualenv node['crytomoneymakers']['venv_path']
    user  'cryptomoneymaker'
    group 'cryptomoneymaker'
    action    :install
  end
end
```

And here is how the cronjobs are created.

```
cron 'funds_fetcher' do
  minute '0'
  hour    '*/6'
  user    'cryptomoneymaker'
  command 'cd /var/cryptomoneymakers/venv/ &&
      /var/cryptomoneymakers/venv/bin/python
      /var/cryptomoneymakers/venv/MillonesApp/manage.py
      fetch_user_funds BotMaster'
end

cron 'tickers_fetcher' do
  minute '*/4'
  user    'cryptomoneymaker'
  command 'cd /var/cryptomoneymakers/venv/ &&
      /var/cryptomoneymakers/venv/bin/python
      /var/cryptomoneymakers/venv/MillonesApp/manage.py
      fetch_ticker_value'
end

cron 'feed_executed_oders' do
  minute '*'
  user    'cryptomoneymaker'
  command 'cd /var/cryptomoneymakers/venv/ && source bin/activate &&
      /var/cryptomoneymakers
      /venv/MillonesApp/backend/cronjobs/feed_executed_orders.py'
end
```

5.3 Python virtual environment

Virtual environments are elements for create isolation. There are tools for isolate the CPU and memory as cgroups, tools for isolating the filesystem like chroot and there are other tools for creating virtual environments for the applications. Python virtual environment is one of the last, it provides isolated environments in terms of installed libraries. Inside a Python virtual environment one can install and run Django 1.6, and in other environment Django 1.7, each one with their respective versions of compatible libraries running. A virtual environment can be used for example for having different environments like: development, QA, staging and production, all in the same machine without worry about broken dependencies. More information about the Python virtual environments ca be found in the official page in readthedocs[6].
To ensure a consistent environment in the project development, all the packages and libraries are installed inside a virtual environment. An important tool inside the virtual environment is pip[7], the Python's package manager. With pip a user can freeze versions of libraries, install news, list the packages installed and much more. Thanks to this, the dependencies of a project are easily seen, for example this project is dependent of the next libraries and versions:

```
(venv)-bash-4.2$ pip list
backports.ssl-match-hostname (3.4.0.2)
bottle (0.12.8)
configobj (4.7.2)
decorator (3.4.0)
Django (1.6)
django-datetime-widget (0.9.3)
iconv (1.0)
iniparse (0.4)
ipython (3.1.0)
meld3 (0.6.10)
MySQL-python (1.2.5)
nose (1.3.0)
numpy (1.7.1)
pika (0.9.14)
Pillow (2.7.0)
pip (6.1.1)
pycrypto (2.6.1)
pycurl (7.19.0)
pygobject (3.8.2)
```

[6]http://docs.python-guide.org/en/latest/dev/virtualenvs/
[7]https://en.wikipedia.org/wiki/Pip_%28package_manager%29

```
pygpgme (0.3)
pyliblzma (0.5.3)
pytz (2015.4)
pyudev (0.15)
pyxattr (0.5.1)
setuptools (12.0.5)
slip (0.4.0)
slip.dbus (0.4.0)
South (1.0.2)
supervisor (3.0)
urlgrabber (3.10)
uWSGI (2.0.9)
virtualenv (13.1.2)
yum-metadata-parser (1.1.4)
```

5.4 Processes manager: Supervisor

Supervisor is a tool for controlling the life cycle of other processes, developed in Python. It acts as a process manager, independently from the operating system process manager like init or systemd, but it is a normal process and its PID is not 1. The normal use case for Supervisor is to start programs and auto restart them if they die. It generates logs from its managed processes capturing their STDOUT and STDERR descriptors. In general it is a very powerful tool which works great in my opinion. More information about supervisor can be found in the official page[8].
The project is using supervisor to be able to restart the process which could crash. It is necessary for the backend processes because they have bugs due the the lack of debugging, and they found sometimes exceptions. Because of the reliability of the project, the backend processes must be all the time running, so Supervisor makes the task of controlling them.
In futures releases of the project, the simple goals that now are achieved via Supervisor will be achieved by systemd. The application is running under CentOS 7, and make sense to use all its tools, mostly the powerful systemd. The appendix shows the configuration files used in the project. **??**

5.5 Git

At the beginning of the project the source code was inside the server and both production and development were using the same versions of it. While

[8]http://supervisord.org/

working in the code, was necessary the use of a revision control tool. For this purpose, Git was chose because it is the most adapted system.
There are many services in internet which offer free accounts to host code, the main one is Github. This service is great but the free version has a limitation: there are no private repositories. This project has been developed thinking in be open source, but I did not want to expose it to everyone until the first release appears.
For this I decided to use my own git server, having different branches: development and production. Like this all the modifications of the code are done in the development branch, and once the code is tested and working, it is merged into production branch. Once in production, a simple "git pull" and restart of the web server will be enough to have the latest version of the code running and accessible by the users.
At the end of the project a total of 155 commits were done.

```
(venv)-bash-4.2\$ git rev-list HEAD --count
155
```

Chapter 6

Front end

6.1 Django overview

6.1.1 Introduction

Django[1] is a free and open source Python Web framework maintained by the Django Software Foundation which follows the model-view-controller (MVC)[2] architectural pattern. Django is designed to offer to the developer rapid development, clean and pragmatic design. It takes care of much of the hassle of Web development, leaving the developer to focus on writing the application without needing to reinvent the wheel.
Is important to understand what means model-view-controller for understanding how a Django application is designed. In the Django interpretation of MVC, the "view" describes the data that gets presented to the user, not how a user see it. So, a view is the Python callback function for a particular URL because describes which data is presented.
Furthermore, it is sensible to separate content from presentation, which is where templates come in. In Django, a view describes which data is presented, but a view normally delegates to a template, which describes how the data is presented. The templates are written using especial language which mixes HTML with simple statements as: if/else conditions, (simple) for loops and variable instantiation.
The controller is the framework itself: the machinery that sends a request to the appropriate view, according to the Django URL configuration. Someone who wants to use a more precise terminology will call Django a MTV framework: model-template-view.

[1]`https://www.djangoproject.com/`
[2]`https://en.wikipedia.org/wiki/Model/%E2%80%93view%E2%80%93controller`

This explanation is simplified in this diagram created by the US National Library of Medicine.

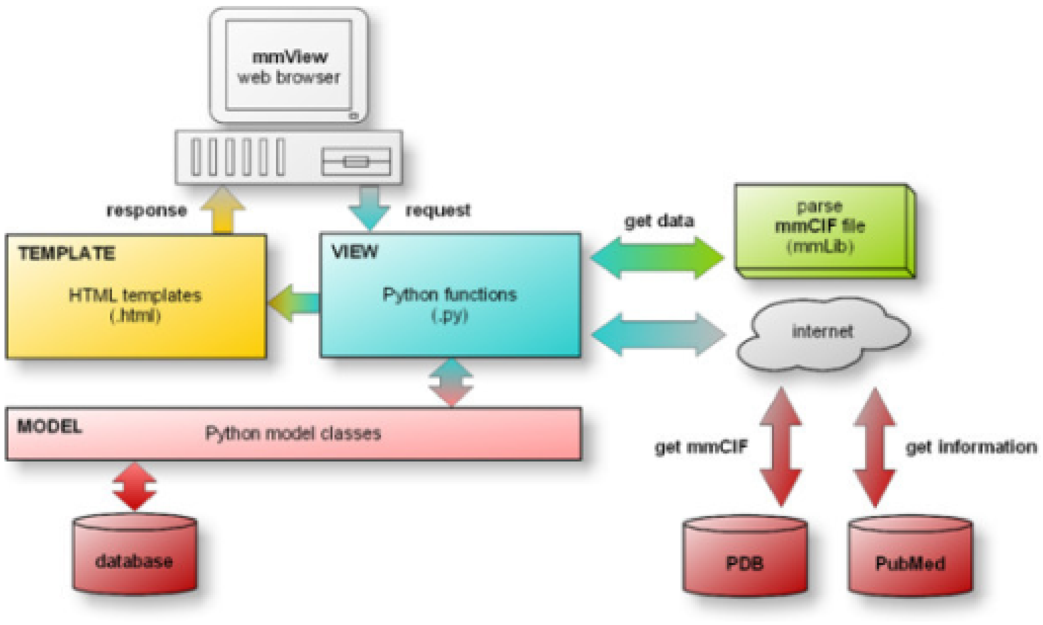


Figure 6.1: Django model-view-controller

6.1.2 Security

Django, as a good framework, is designed to avoid the most common web security issues. By default it tries to protect your site against:

- Cross site scripting (XSS)[3]: the templates are escaping most of the dangerous HTML characters.
- Cross site request forgery (CSRF)[4]: Django checks for a nonce in each POST request of a user, making impossible the replying of a package. But if the attacker intercepts the user cookie and the secret key, he could be able to modify and reply a package.
- SQL injection protection[5]: by default the queries to obtain objects stored in the database are not written by the user using SQL, but as simple objects and filters notation. This ensures that a novice developer will not introduce weak SQL queries.

[3]https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
[4]https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)
[5]https://www.owasp.org/index.php/SQL_Injection

- Host header validation[6]: Django validates the HTML host headers and ensures to match a list of enabled host names defined in the settings.
- SSL/TLS[7]: it allows to easily use secure socket layer encryption.

6.1.3 Structure of a Django project

Django does not have a special folder structure for organizing the projects, but most of the community projects follow the next folder structure.

- Project name
  - App name
    - models.py
    - urls.py
    - views.py
    - forms.py
    - templates
  - static
  - settings.py

All the project is contained inside one folder named as the project; inside it, there is one folder per each application contained in the project. Each file inside an application accomplish a specific and well known function. The next sections are giving a brief overview of each file and function.

Models

The file models.py contains the definition of the objects of the application, which will be stored into the data base. Each object is composed of attributes and methods for describing its behavior. The attributes of each class are strongly typed, and properties as length of default value can be specified. More information about the models can be founds in the official documentation[8].
To illustrate the concept of a model, an example of one part of the sliced order object is reported.

[6]https://www.owasp.org/index.php/HTTP_Request_Smuggling
[7]https://en.wikipedia.org/wiki/Transport_Layer_Security
[8]https://docs.djangoproject.com/en/1.8/topics/db/models/

```python
class BtceOrder(BaseOrder):
    '''
    Class containing attributes and methods to represent a valid
        btc-e order

    Attributes:
    buysell: String with possible values 'buy' || 'sell'
    pair:   Valid btc-e pair
    amount: Quantity of coins to exchange
    status: Can be 'created' || 'started' || 'canceled' || 'executed'
    btceid: Id returned by btce.
    '''
    pair = models.ForeignKey(Change)
    amount = models.FloatField()
    price = models.FloatField()
    buysell = models.CharField(max_length = 4) # allowed values:
        'buy' or 'sell'
    btceid = models.IntegerField(null = True)

    @classmethod
    def create(cls, pair, user, buysell, amount, price):
        order = cls(pair=pair, amount=amount, price=price,
            buysell=buysell,
        user=user, status='created')
        order.save()
        return order
```

This model represents a btce order. It inherit from base order, which is the template used by the rest of the orders. The firsts lines are used to document the object, using the PEP 0257 docstring conventions[9]. After, 5 attributes are defined: 2 floats for storing amount and price of an order, 2 strings for containing the identifier returned by btc-e.com when creating the order and type of the order. The fifth attribute is a foreign key to a Change object, which is the identifier of the pair to trade with.
The create method is used as constructor method. So, it receives all the attributes needed to create the object, initialize it, and save it into the database.

URLs

Django proposes to use a clean schema for the URIs and really encourages users to do that[10] even if does not put any restriction. The file urls.py is where the mapping between the URL requested by a user and the code executed

[9]https://www.python.org/dev/peps/pep-0257/
[10]https://docs.djangoproject.com/en/1.8/topics/http/urls/

by the application is done. Basically it analyzes the URL requested by the user, using regular expressions and executes a view function for each match, if the URL does not match any rules, the web server will return an HTTP 500 error.
A reduced example of a *urls.py* file is shown below, where the URL will try to match the patterns for a sliced order or paired order.

```
urlpatterns = patterns('',
url(r'^slicedorder/$', views.slicedOrder, name='slicedOrder'),
url(r'^pairedorder/$', views.pairedOrder, name='pairedOrder'),

)
```

These lines are mapping an URL containing *slicedorder/* or *pairedorder/* to the correspondent function for processing the request. The third parameter, 'name', is used for an abstraction inside the Django code. The URL matching settings is nested, so each application might have a *urls.py* file. This structure is not mandatory and the nesting levels can be as deeper as the developer wants.

Views

One view is a function that takes web requests and returns web responses. A response can be a rendering of a template, a redirection, an HTML error, etc. In most cases a view takes the request from a user, handles it, creates dynamic content and renders it through a template. More information about Django views can be found in the official documentation[11].
To illustrate the concept of a view, the next example shows a very short one.

```
@login_required
def index(request):
  '''This view represents the main wallet view'''
  sk = str(request.user.userprofile.btce_secret_key)
  ak = str(request.user.userprofile.btce_key)
  info=get_info(sk, ak)
  return render(request,'wallet/main.html',{'info':info['return'],})
```

The first line of the code above contains an '@', it represents the use of a Python decorator[12]. A Python decorator is a function which returns true or false values, and the execution of the next sniped of code is conditioned to the result of the evaluation. So, in this case, using this decorator is ensured that the function is executed only if the request comes from a logged user.

[11]https://docs.djangoproject.com/en/1.8/topics/http/views/
[12]http://thecodeship.com/patterns/guide-to-python-function-decorators/

The function gets 2 attributes from the user profile: API key and secret key. Then, it gets information about the current status of his funds, using a method to query btc-e.com, and returns an HTTP page through rendering the template.

Forms

All the forms that an application has are defined in *forms.py*. Forms in Django are very powerful and customizable, they can be defined field by field or inherited from a model. When inheriting from a model, a form will have one field per attribute of the class, making super fast the form definition.
This project is using many forms and experimenting with them, overriding its internal methods and integrating external widgets. A more detailed explanation about each form used by the project can be found in the correspondent section.

Templates

The folder templates contains all the templates used by the the application. A template is a file which contains HTML and a special template language[13] which is processed by the Django engine and rendered as HTML page to the final user. So, each HTML file represents a page that a user might see.
The Django template language is a mini-language which has only few basic functions, it can do simple loops and comparisons over data as well as including files and inherit from other templates.

Settings

A Django settings file contains all the configuration of Django. It can be extended by the developer to incorporating new values. Here everything related with Django is configured, the main settings are listed and explained below.

- Base directory: route to the directory.
- Secret key: password used to provide cryptographic signing, it should be unique and unpredictable.
- Debug: boolean to set the debug mode of the application at on or off.
- Allowed hosts: list of the valid addresses or names of the project.

[13] `https://docs.djangoproject.com/en/1.6/topics/templates/`

- Installed apps: list containing the names of the applications used in the project.
- Databases: setting used to connect to the database, including host, user, password and engine.
- Location: setting where is defined the language, time zone, etc.
- Static: contains the path to the folders with static files.
- Logging: defines the different logging handlers used in the application[14].

Static

The static folder is where an application stores its static contents like images, Javascript, CSS, etc. Using of a static folder in each application rather than one globally, ensures the correct isolation of an application and helps to reuse the application in other projects.

6.2 Implementation

6.2.1 Base template

The base HTML template defines the main structure of the web application, all the other templates included in the project are inheriting from this one. The whole code of the template is reported here.

Listing 6.1: Base template.

```
<!DOCTYPE html>
<html class="no-js" xmlns="http://www.w3.org/1999/html">
    <!--<![endif]-->
<html lang="en">
    <head>
        <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
        <meta charset="UTF-8">
        <!--<meta name="Cryptomoneymakers" content="Trading platform
            built on top of btc-e">
        <meta name="keywords" content="bitcoin trade btc-e">-->

        <title>{% block title %}{% endblock %}</title>
        {% load staticfiles %}
        <link href="{% static 'css/simple.css' %}" rel='stylesheet'
            type='text/css'>
```

[14] https://docs.djangoproject.com/en/1.8/topics/logging/

```
       <link rel="stylesheet" href=
           "https://maxcdn.bootstrapcdn.com/
           bootstrap/3.3.5/css/bootstrap.min.css">
       <link rel="stylesheet" href=
           "https://maxcdn.bootstrapcdn.com/
           bootstrap/3.3.5/css/bootstrap-theme.min.css">
       <link rel="stylesheet" href= "//netdna.bootstrapcdn.com/
           font-awesome/4.2.0/css/font-awesome.min.css">
       <script src="https://ajax.googleapis.com/ajax/
           libs/jquery/1.11.3/jquery.min.js"> </script>
       <script src="https://maxcdn.bootstrapcdn.com/
           bootstrap/3.3.5/js/bootstrap.min.js"> </script>
       {% block header %}{% endblock %}
    </head>
    <body>
       <div class="row">
    <div class="grid_12">
           {% include 'main_nav.html' %}
         </div>
      </div><!-- end row-->

       <!-- main content area -->
       <div id="main" class="wrapper">
           {% block body_main %}{% endblock %}

       </div><!-- #end div #main .wrapper -->

  <nav class="navbar navbar-default navbar-bottom" role="navigation">
    <div class="container" style="text-align:center">
      <h5> Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com
          </h5>
    </div>
  </nav>
    </body>
</html>
```

The template starts defining the kind of document (HTML), the language used by the page (English), a header section, a body section and close the HTML tag to mark the end of the document.
The header section defines meta information describing the encoding set used, the name of the page and keywords used by the browsers and search engines to render and index properly the page. The page name appears enclosed by the two statements *% block title %% endblock %*, this is not HTML but Django template language markup. These statements define a block without content inside, which is used by te other pages to set the title of the page that are rendering, just using the same syntax and setting the title.
After appears the statement *% load staticfiles %*, which is used to tell to the

Django rendering engine that this page will use static elements. The next 6 lines are inclusions of the bootstrap CSS theme and Javascript functions, also the *simple.css* file which was the style used at the very beginning of the application's development and is currently deprecated.
After the header, comes the last section: body. This HTML section is where is located all the content visible by the end user. It defines 3 subsections.
The first one is a row of width 12 (this means that will occupy all the possible width of the page) and includes the file *main_nav.html* which is described in more detail in the next sub section.
The second element is a row with other block called body_main having a inclusion Django tag inside. Here is where all the applications will put their content.
The third element in the base template is the footer, where information about the author is displayed. The footer could be extended adding links to the documentation, contact API, etc.

Navigation menu

The main navigation part is the menu that appears at the top of the web application. This menu serves for navigating through the different applications of the project. The design has a gradient color from white until gray that gives a sensation of cleanliness and space. In the left side of the menu is situated the logo, which is a dollar bill for representing the American spirit of getting rich starting from one dollar.
A no logged user ca not see anything else because the navigation between the sections is only displayed for a logged user. However, a logged user will see three drop down menus located at the left of the bar: Wallet, Orders and User name. The first one shows, after the user click, all the sections of the wallet application, the same behavior occurs in Orders. The last one shows the user name and when clicking, a drop down menu gives the possibility of navigate through his settings and to log out.
The file containing the actual code is listed in the appendix 10.3.1.

6.2.2 CSS

Bootstrap 3

Bootstrap is a free and open-source framework for creating websites and web applications[15]. It contains HTML- and CSS-based design templates for

[15]URLhttp://getbootstrap.com/

typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. It aims to ease the development of dynamic websites and web applications.
It was originally named Twitter Blueprint, and developed by Mark Otto and Jacob Thornton at Twitter as a framework to encourage consistency across internal tools. It was renamed from Twitter Blueprint to Bootstrap, and released as an open source project on August 19, 2011. It has continued to be maintained by Mark Otto, Jacob Thornton, and a small group of core developers, as well as a large community of contributors[16].

Reasons for using it. Bootstrap has been chosen because it simplifies a lot the task of the web development in terms of page structure, disposition of the elements and mostly in terms of compatibility between the different kind of devices.
The version used in the web application is 3.0, which adopts a mobile first design philosophy, emphasizing responsive design by default. This make it perfectly suitable for building a new web application for a non web developers. Other benefit is that is widely used across internet and has a large community with tons of questions answered of how achieving the objectives. At this point of the project development, the decision of using Twitter Bootstrap seems to be the correct one.

Icons

The web application uses icons to represent the currencies and to make more pretty the navigation menu of each sub application. Nowadays is a common pattern between the web developers to use icons as a mechanism of universal communication and as creation of visual interest. Due to these two reasons, the application uses icons: trying to capt all the interest of a user. Furthermore, the use of icons does not influence the time of loading a page.

Coins. The table displaying the funds, for example inside the simple order section, includes a different icon to represent each of the currencies. Their style is not completely homogeneous because it is difficult to find a set of this kind of icons in internet for all the available currencies. Some currencies like the NovaCoin does not have an official icon yet. However the result looks very nice achieving certain degree of uniformity regarding shape and size.

[16]URLhttps://en.wikipedia.org/wiki/Bootstrap_%28front-end_framework%29

Navigation menus. The icons used for the navigation menu are homogeneous in terms of style. All of them are part of the Font Awesome project[17]. This projects is an open source set of clear fonts and icons very popular between web developers and designers. At the moment of writing this text it had 35.622 starts and was forked 5.822 times in Github.
Each section inside the applications is mapped into an icon, most of the times the icon describes truly the function of the section, but in a few exceptions, was not possible to find an icon in the Awesome icons set which was describing the behavior. The use of the icons makes the menus look very nice without adding a difference in the time used to render the page.

6.2.3 Charts

The project is displaying some graphs and charts with information regarding the number of orders done by a user or the total amount of money that a user has. There are many frameworks in internet offering the possibility of rendering graphs, open source and with proprietary licenses.
Some examples are the D3js graphs[18] which is a collection of very powerful and awesome graphs. Other possible choice widely used is SigmaJs[19]. Another one is to render the graphics in the backend using some tool like GNUPlot[20] and displays the result as an image.
In favor of all the options, Google chart was chosen[21]. This chose was done because this library is very simple to use, completely open source and the documentation is very good, including examples of each kind of chart and description of each possible attribute. The Google charts have the disadvantage of being not as powerful as other libraries, but enough for the purpose of this project.
The way that the project is using them, is embedding the Javascript code in each page that displays a chart. This approach is not bad but could be better using third party projects which integrates the charts with Django applications in a more elegant and flexible way, for example: Django-charts[22].

[17]`https://github.com/FortAwesome/Font-Awesome`
[18]`http://d3js.org/`
[19]`http://sigmajs.org/`
[20]`http://www.gnuplot.info/`
[21]`https://developers.google.com/chart/`
[22]`https://github.com/rhblind/django-gcharts`

6.2.4 Disclaimer

The HTML, CSS and Javascript used in the project probably do not have the best structure and are not following the bests practices in web development. However this project is not about web development, the fact of having web languages is a necessity rather than a feature. Nowadays the development of a desktop application oriented to normal public does not have sense, because everything is based on the web and is designed to be usable in desktop computers, laptops, tablets, phones, etc. Anyway the development of this part of the code is trying to be simple and effective to be able to extend it easily without starting from scratch.

Chapter 7

Application insights

7.1 Introduction to the chapter

This chapter contains the enumeration and description of the Django application and its components. It is the most important chapter for understanding the whole project, because the applications explained here have the structure used by the final user to interact with the system.
The chapter is organized using one section per application. Each application is subdivided in logical parts, which normally represents a different web page. There are some exceptions to this, like the logical parts explaining components which do not have visual representation.
The user will be guided through the different parts of the logical units: models, views, forms and templates.

7.2 Users

The users application is the one which manage the creation, deletion, and customization of the preferences of the users. For the creation of a new user, there is a registration section which is disabled until the application will be in production. It is disabled for protecting the system against malicious users or people who just want to look around.
The process of login into the web application is done via the login section where a registered user, using his user name and password, will be able to start using the application. After finishing to trade, he will log out via the log out section.
The user can change his password or his btc-e API keys via the setting section.
The mapping between the URL requested by the user and the function called

is done by the urls.py file. The content of this file can be found in the appendix 10.3.3.
The next subsections explain with high level of detail the components of the users application.

7.2.1 Settings

The user's settings section is the part of the application where a user can see and modify his preferences. Currently it is possible to operate with the user password and the user btc-e API keys. These are two very basic, but needed, settings. In future releases of the application this section will be extended to manage the user email address, the currencies that he want to see and trade with, a maximum limit of money traded per day, etc.
The details regarding the implementation of each available option is explained below, first for the view function and after for the template.

Views

Info. In the info section a user will see his main information: (public) API key, the date of registration and last login. This section is indeed to give an overview of the user settings.
The view function is the shortest one of all the application, it does incredibly nothing but renders the template, without passing any arguments to it. This is because in Django the user object is always passed to a template, and the fields that are displayed here are actually part of the user model. The code reference can be found here 10.3.3.

Change password. Change password view is where a user can change his password for login into the application. As in most of the web applications, a user must introduce his current password and two times the new password to be able to change it.
Asking the user ot introduce his current password ensures that he is who is using the application, and asking him to repeat two times the new password makes sure that the user does not make a typographic error. After this information is sent, it is validated to ensure that the constraints of introducing the correct current password and the equality of the two fields for the new password are fulfilled.

Change API keys. The change API keys view is where a user can modify his btc-e.com API keys. This section is very important because from time to time btc-e.com revokes all the keys forcing the user to change them.

The view displays two fields for introducing the public and the secret key. At the moment, this information is not validated against the btc-e API to test if the keys are valid.

Templates

Info. The info template shows a customized welcome message for the user. After, the information mentioned above is displayed clearly inside a table without borders, and overriding the padding between the cells to get a nice visual aspect. The title of each item uses bold letters. The code for this template can be found here .

Change password. The template in charge of rendering the page for changing the password is pretty straightforward: it checks if a success or error message, containing result of a previous operation, was passed as an argument, displaying it with green or red colors. Finally, it renders a form with the three fields for changing the password.

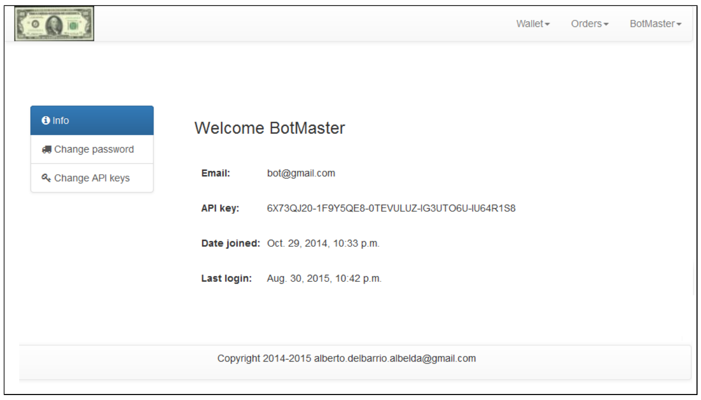
Change API keys. The code for the change API keys templates is pretty similar to the one explained above with the only difference in the names and quantity of fields.

Forms

The users application defines two forms: change password and change API keys.
The change password form is special because it contains a method for validating the password field: *clean_newpass2*. It checks if the two introduced passwords are the same using the Django internal mechanism for validating fields[1].
Change API keys is a simple form with two fields for setting both the public and private API keys of the user.

[1]`https://stackoverflow.com/questions/7948750/custom-form-validation`

Screenshots



Welcome BotMaster

Email: bot@gmail.com

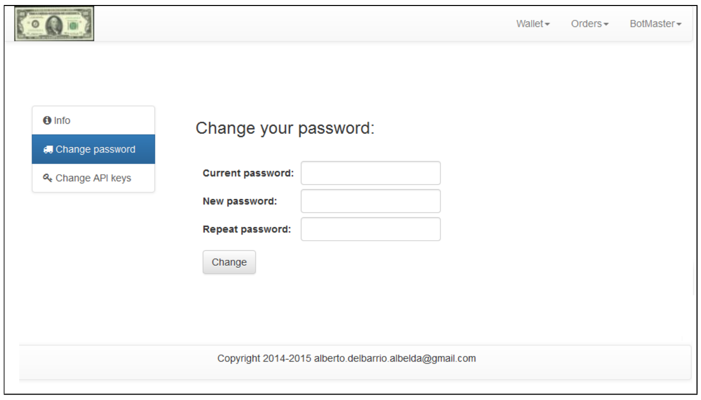
API key: 6X73QJ20-1F9Y5QE8-0TEVULUZ-IG3UTO6U-IU64R1S8

Date joined: Oct. 29, 2014, 10:33 p.m.

Last login: Aug. 30, 2015, 10:42 p.m.

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.1: Info page.



Change your password:

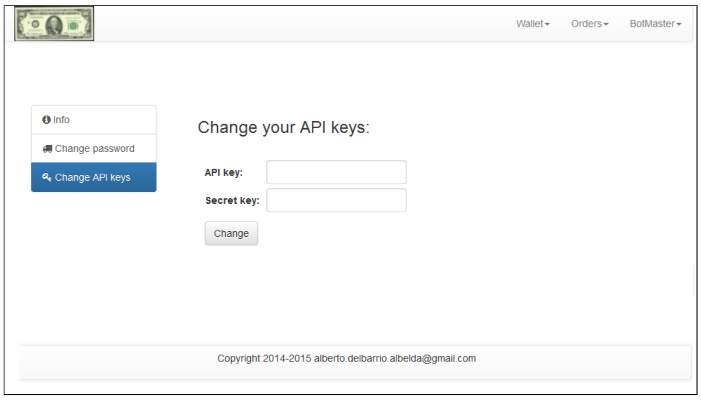
Current password:

New password:

Repeat password:

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.2: Change password page.



Change your API keys:

API key:

Secret key:

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.3: Change API keys page.

7.2.2 Log in

The log in section is the first page that a user sees when entering in the application. It handles the process of authenticating a user against the application. To be authenticated the user must provide a registered user name with the corresponded password. If the user provide valid details, it will be able to use the application until the logout. If the log in details provided by the user are not correct, he will be able to try again as many times as he wants. This behavior has to be changed when the application will go into production to avoid possible attempts to obtain the password of a user. This probably is easy to do using a third party application like django-axes[2].

View

The user log in view handles the part of the user authentication against the Django application. For authenticating it uses the authenticate method provided by the *Django.auth* module, which returns true if the details provided are correct and false if not. After, the code checks if the user account is available. For the moment all the accounts are available, but this could be used in the future to ban users who do not respect the rules. If the attempt to log is not successful, the user is redirected to a page showing an error and the attempt is logged.

Template

The login template for rendering the page creates a form with two fields: one for the user name and other one for the password. This is not the same structure than in other parts of the application where the form is previously created in the forms.py file and just rendered in the template. This template still creates a form manually because this was one of the first parts developed in the application, and how to work with forms was not so clear yet. This will be changed in favor of a form in forms.py in futures releases of the program.

[2] `https://github.com/django-pci/django-axes`

Screenshot

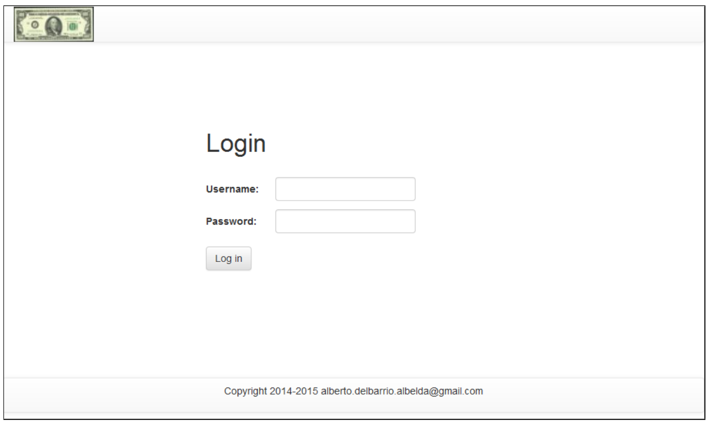


Figure 7.4: Log in page.

7.2.3 Log out

The log out section is very little, only composed by a view function. When a user requests this function via the log out button located at the top navigation bar, a call is done to the logout method. The *Django.auth* module revokes the temporary data created to mark the user as logged.

7.3 Wallet

Wallet application is the electronic wallet of a user: it shows the balance of the user account, list high detailed reports of the orders finished and allows the user to get the orders history as well as being able to filter these results in the last transactions.
When a user first log in into the system, he will navigate to the main view of Wallet where he will see an area chart with historical data displaying the total amount of his wallet expressed in USD. When scrolling down he will see a bar graph showing the number of orders done for each type. Thanks to this graph a trader can easily understand how is the status of his account.
Then, he can navigate to the report section to see which orders are completed and the results of the operations. In this view the user will see five tables, one for each kind of order, each one showing the reports for the last five orders. Here the user will explore each order to see how it finished.
Before start trading he might like to see the amount of each currency that he owns to prepare the strategy of the day; the application shows this information in the Funds section.

The application also allows a user to query the orders or transactions realized in the past. So the user can navigate to the correspondent view and make a query filtering the available data, being able to select between a range of dates, the amount of results or the trading pairs.
These five features are logically separated in six different components of the application. The next sessions describe in high detail the implementation of each one of them.

7.3.1 Summary

Summary is the first view which appears in the wallet menu and is designed to give a fast overview on the status of the wallet of an user. The main content is displaying three different elements.
The first element is a table which contains the useful information returned by the API method *get_info*. The data displayed in the table are: the number of active orders in btc-e, the number of opened transactions active at the moment in btc-e and the rights that have the API key used by the user.
The second element is an area chart showing the total amount of money owned by the user during a period of time of two months.
The third and last element contains a bar chart showing the total number of orders of each possible type realized by a user.

View

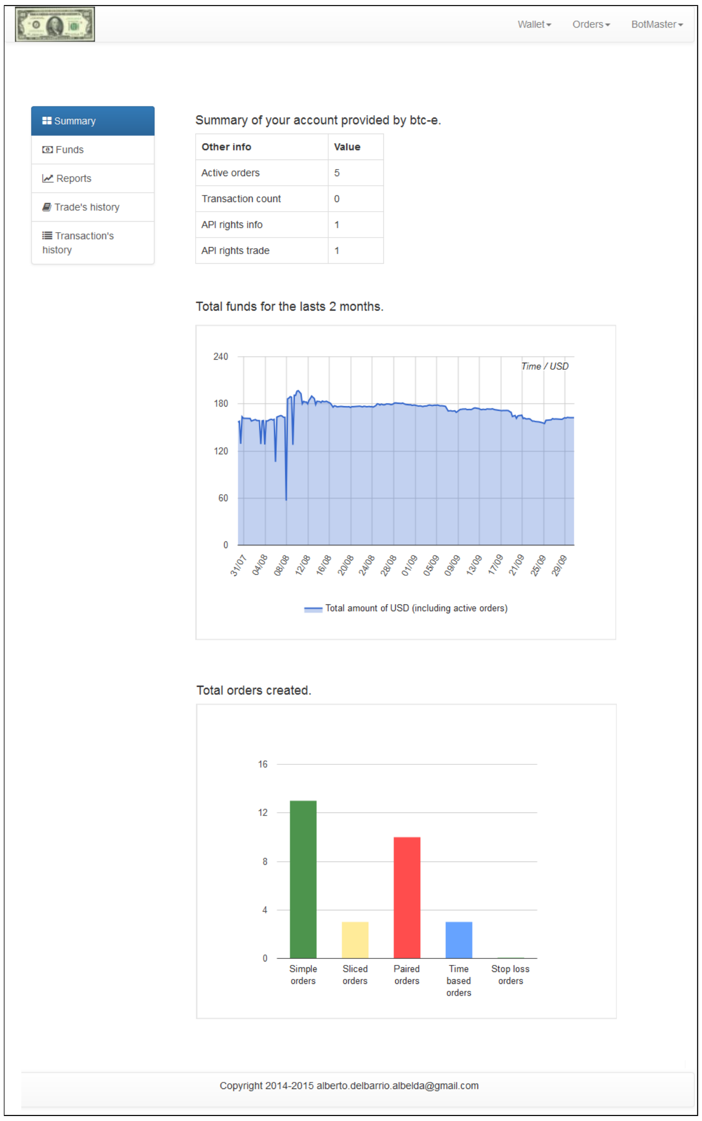
The summary view has the code to generate the 3 different elements. The code is divided in three blocks, each one of them fetches the information for an element and adds the results to the context dictionary.
For getting the information about the API key rights, transactions and orders opened, only two lines are needed: one for querying the btc-e API and other to store the result.
The code for displaying the graph of the total amount of money owned by a user, is composed by just three lines: the first one queries the database to get the lasts 200 orders, the second one formats the data to make it understandable by the google chart javascript function and the third one stores the results into the context dictionary.
The last section will display a bar graphic representing the amount of orders of each kind created by a user. To get this information from the database, one query per kind of order is necessary, which returns all the orders for each kind. However only the size of the set is needed. This part of the code is not so good because to get all the orders is an expensive operation in terms of memory, and it is used only to calculate the length of the vector. In the

future has to be improved.
The code itself can be found in the appendix 10.3.4.

Template

The code of the template used to render the summary is simple and clear: it displays three different components. There are three "if" conditions, checking for the existence of each component, the first one renders the table resume, the second one contains a Javascript function for rendering the graph of the total amount of money, the third one contains a similar script which will render the graphs showing the number of orders. Inside the first script there is "for" loop printing the data collected in the view and an API call to the Google API for rendering the graph. This two scripts set the style variables to get different results regarding the kind of graphic, the color of the bars, the border of the background, etc.

Screenshot



Summary of your account provided by btc-e.

Other info	Value
Active orders	5
Transaction count	0
API rights info	1
API rights trade	1

Total funds for the lasts 2 months.

Total orders created.

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.5: Summary page.

7.3.2 Funds

Funds is central place where a user can see in real time the content of his wallet. In only one table a lot of useful computed information are displayed. It contains one row for each coin, currently seven (PPC, USD, LTC, NVC, NMC, BTC, EUR) and three columns: Available, In orders and Total. Each column is subdivided in two columns: - and USD. The most important feature

of this table is the fact that shows the amount of each coin in USD, and this is super useful to get a fast overview of the volume that each coin represents. To obtain the amount each coin is translated into USD, the last traded value in btc-e is used. So it is in real time. Nowadays the translation of the value into USD is very easy to do, because all the currencies supported by btc-e can be changed directly into USD. In the first stages of the development was not like this, and currencies like XPM could only be converted to BTC. Those days the translation of the values was a challenge to face.
The result is a nice and compact table that shows very useful information in a simple way.

View

The view function contains the code for fetching and converting to USD the funds of a user.
First the function gets the active funds of the user in real time querying the btc-e API. Second it converts the funds to USD using an auxiliary function: *convert_funds_to_usd(funds)* which receives a dictionary with currencies and translates this amount to USD getting the current price of each ticker. This function just loops over the dictionary of the funds, and calls *convert_to_usd(coin, amount, tickers=None)* which does the real calculus. The last function receives optionally a tickers dictionary, and if it does not receive it, queries btc-e to get it. The implementation is getting only one time the tickets and reuses it to convert all the coins to USD reducing like this the total amount of queries to btc-e API and like this reducing the total time needed for render the page.
The function continues getting the funds in real time used by the active orders of the user. Doing this process is not trivial, because once having the list of the active orders, the amount of each coin has to be extracted from the name of the pair, so can be of two currencies. For example for an order of buying 2 BTC at 200 USD the information available is the name of the pair: btc_usd. Studying the structure of the available pairs in btc-e I have figured out that if the order is a sell order, the money invested will correspond to the first coin appearing in the pair, and if the order is type buy, the amount invested will correspond to the second coin in the pair. The example reported is for a buy order, so the total currency invested of that order is 200 for the second coin, that is 200 USD. The code that implement this algorithm is wrapped into *get_funds_in_active_orders()* function listed below.

Listing 7.1: Get funds in active orders.

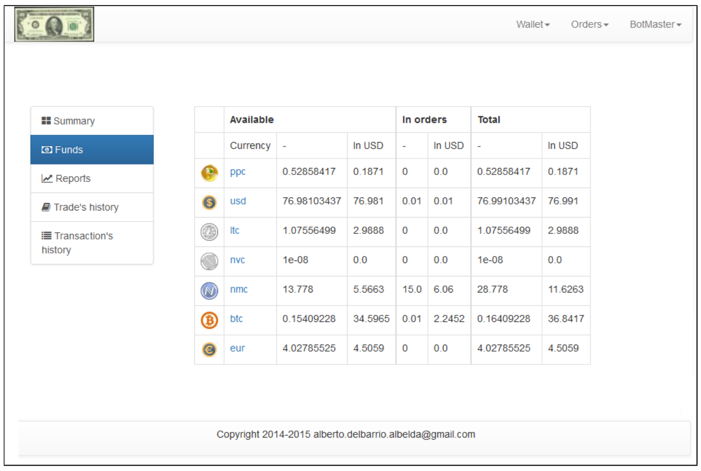
```python
def get_funds_in_active_orders(sk, ak):
    '''
    This function gets the amount of currency present in orders
        active.
    It returns a dict containing the values for each currency.
    '''
    #Add the funds in active orders:
    ao = get_active_orders(sk, ak)
    funds = {}
    if ao and ao['success'] == 1 and len(ao) > 0:
        for k, v in ao['return'].iteritems():
            if v['type'] == 'sell': #if sell, the currency owned is
                the first in the pair
                funds[v['pair'][0:3]] = v['amount']
            else:
                funds[v['pair'][4:7]] = v['amount']
    return funds
```

Once the amounts are calculated and stored into a dictionary, this dictionary gets enlarged adding all the possible coins which where not presented in available orders, with an amount of 0. This is done to make easy the building of the table when rendering the template.
Then, the data of this dictionary is converted to USD.
The last step consist in summing the two dictionaries to obtain the total amount of each currency, and how much does it represent in USD.
The function is awesome but it has also disadvantages: it is slow because performs several queries to the btc-e API. It has been already be tunned to reduce the number of queries, but the main bottleneck resides in the method of *get_ticker()* that query the btc-e API.

Template

The template that generates the HTML page is straightforward to understand: it just loops each one of the three dictionaries for adding the information contained into the table. A careful reader looking into the code will notice that actually it is not one table but three, one per each dictionary. This implementation has been chosen in favor of the use of only one table, because using the Django template language it is very difficult to iterate over the keys of 3 dictionaries at the same time. This solution is elegant because keeps the code simple.

Screenshot



	Available			In orders		Total	
	Currency	-	In USD	-	In USD	-	In USD
	ppc	0.52858417	0.1871	0	0.0	0.52858417	0.1871
	usd	76.98103437	76.981	0.01	0.01	76.99103437	76.991
	ltc	1.07556499	2.9888	0	0.0	1.07556499	2.9888
	nvc	1e-08	0.0	0	0.0	1e-08	0.0
	nmc	13.778	5.5663	15.0	6.06	28.778	11.6263
	btc	0.15409228	34.5965	0.01	2.2452	0.16409228	36.8417
	eur	4.02785525	4.5059	0	0.0	4.02785525	4.5059

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.6: Funds page.

7.3.3 Fund

The fund section was created to give to the user details about how much of one coin owns. Using this information a user can see easily the history of a particular coin.
To achieve this goal, the main view shows an area graph containing the amount owned of a currency during the period of the last two months. This is not real time data, it comes directly from the database.
This data his collected by the part of the backend used the get the user funds 8.4.1.

Model

Funds is the model for storing the amount of a coin owned by a user in a certain time. It is named in plural because is part of a legacy code, in some moment, it has to be renamed to Fund to follow the conventions.
The fields of this model are self explanatory: currency is the name of a coin, amount is the amount owned by a user, datetime is the date and time when the data was taken and user is a foreign key to the Users model.
There are two exceptions in the currency name: a part of the name of each currency, it can take the value of *tav* to store the amount of each fund returned by the btc-e API and *ttl* to store the total amount of a currency counting also the amount in active orders. These names are not self explanatory because they use only three letters to describe its content. These names have been chosen to continue being coherent with the symbols of each coin.

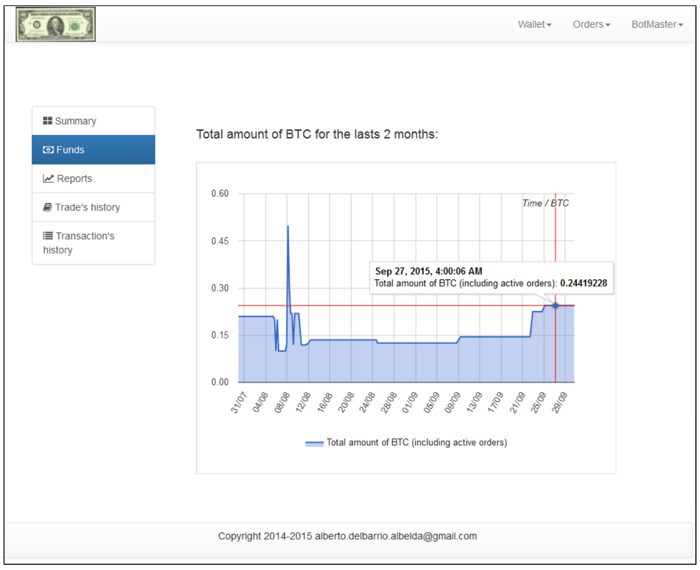
View

The fund view gets the data of the fund and renders the page to the users. The code queries the database to get the historical data of a certain fund (received as a parameter of the function), builds a dictionary with the data and returns calling the render function.

Template

The code of the template is similar to the section that renders a graph in the summary page: it adds the Javascript function to call the Google API and loops over the generated dictionary to display a tuple like (time, amount).

Screenshot



Total amount of BTC for the lasts 2 months:

Sep 27, 2015, 4:00:06 AM
Total amount of BTC (including active orders): **0.24419228**

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.7: Fund page.

7.3.4 Reports

Reports is one of the most important section of the whole project, it shows a lot of information about the orders executed. A trader needs to review constantly the orders executed to understand how they finished, and try to analyze how could have had more benefit. Btc-e only shows to the user, if an order was completed, the same information that he introduced to create the order. This is not enough for someone that wants to win money trading with currencies. The reports section has been developed to provide a user

the information that he need, without having to realize manual calculus.
The project allows to create different kinds of orders, and each one needs different data to be filled and can be finishes in different ways. Due to the diversity of orders was not possible to design just a single format for showing results, so the final view for reports is divided in five sections, one for each kind of order.
The first section displays the results of the lasts five simple orders executed. These results are simple showing if an order was completed or canceled.
The next report shows information for sliced orders. This kind of order is way more complicated than the previous one, and this can be observed looking at the size of the reports. For each of the lasts five sliced orders executed there are two tables: the first one shows the values that the user had introduced to create the sliced order as well as the final status: executed or canceled. This table will remember to the user about the order with a fast overview of how much was traded and between which boundaries. The second table details each and every of the sub orders created by the sliced order, giving information about the amount that was traded with, the price and the final status.
For example, a user had created a sliced order of buying 3 BTC between 100 and 300 USD, slicing the order in three sub orders. The main table will show this information and the second table will contain three rows (one per order) displaying 1 BTC 100 USD, 1 BTC 200 USD and 1 BTC 300 USD respectively. After looking at this information one user can evaluate if the order was sliced properly or if it could be adjusted with other boundaries to get more benefit.
Scrolling down in the page a user can observe the reports for the time based orders. This kind of orders are very similar to the simple orders, just with one difference: they will expire in a certain time if the order was not executed. The report shows the final status of the last five time based orders (canceled, executed or expired) and the expiration date and time. For example a user that left a time base order before go to sleep, can see the next day if the order was finally executed, of if it was expired.
The next report shows the paired order, this is the most fancy report. The paired orders were created to enable a user to open and close a position for a certain trade and let him know automatically how much benefit he obtained. This is one of the most commons operations that a trader realizes in one normal day. Without this calculus, a user which has been playing in the btc-e exchange for a while will not know which orders were profitable and which ones were not. The report for the paired order shows the lasts five paired order executed and calculates the benefit obtained when opening and closing the position, including also the fees of the transactions payed to the

exchange market. This reports are a nice feature that most of the traders want to have in their trading management system.
The last report shows the final status for a stop loss order. This report is needed for the trader to understand if the order was traded with the price that he would like, or as the opposite, the order was traded using a unfavorable value, that is limiting his loss.
The next sub sections are discussing how all of these reports are implemented.

View

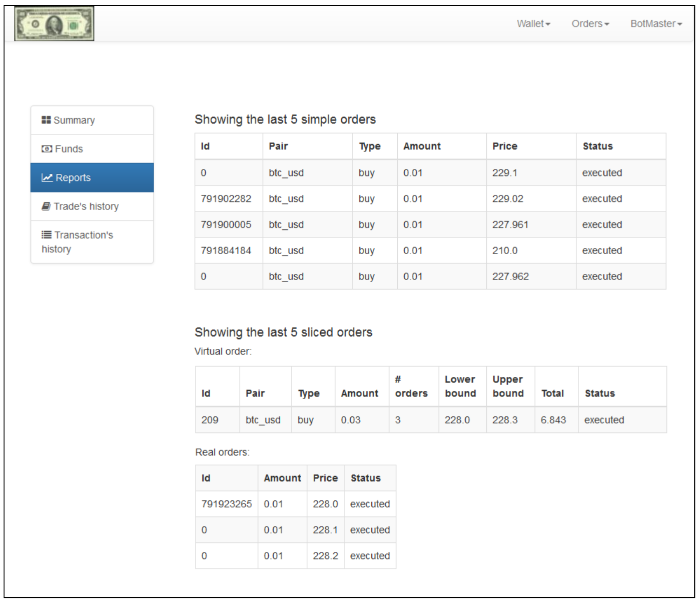
The view of the Reports section is in charge of gathering the information relevant for each kind of order and of transforming this information. The code is longer than in other views, more than 80 lines, but it is easy to understand reading the section above.
For a simple, time base and stop loss orders, it extracts the information from the database without doing any formating or calculus. From this information, it takes only the lasts five results or, if less than five orders has been realized, it will take all the finished orders.
The sliced orders are different, because for displaying the report, not only the last five sliced orders are needed, but also all the suborders that compose each one of those. A part of getting the suborders, the information has to be prepared before be rendered by the template. To achieve this, after having the lasts five sliced orders (or less), there are two nested loops which are filling the dictionary "sliced" adding the information for the global sliced order and part of the information of each one of the suborders. The result is a Python dictionary with all the data structured for being rendered using two loops by the template.
The report for the paired order also transforms the information extracted from the data base. Actually it calculates the benefit obtained with a paired order. A developer who cares about having an optimal code in terms of memory and CPU consumption will see this part as a waste of CPU, because these calculus are done each time that a user see his reports. Seems to have sense to add a field into the paired model named "benefit" and fill it in the moment of marking the order as executed. This is true, but the reason why this code remains in to the view is to give more time to debug the algorithm and evaluate if the results given by the program are correct. If a non correct result is observed, the developer can just change the code in the view and will see the value updated. In some time this code will be incorporated in one method of the paired order class.
Continuing with the possible improvements, has to be notice that the approach of extracting all the orders realized by a user to after get only the

lasts five, is not good: it wastes memory and bandwidth while communicating with the database. However I could not find a better approach using the functions that Django provides.

Template

The template renders the reports page. This code is, as the other templates, quite simple. It checks if there is information for the orders. If there is no information for some kind of orders, a sentence will be displayed in the position where the reports for that order should go adverting the user that there is not data for this order. If there is information, in the dictionary, it will loop over it displaying in tables the contents of each dictionary.

Screenshot



Showing the last 5 simple orders

Id	Pair	Type	Amount	Price	Status
0	btc_usd	buy	0.01	229.1	executed
791902282	btc_usd	buy	0.01	229.02	executed
791900005	btc_usd	buy	0.01	227.961	executed
791884184	btc_usd	buy	0.01	210.0	executed
0	btc_usd	buy	0.01	227.962	executed

Showing the last 5 sliced orders

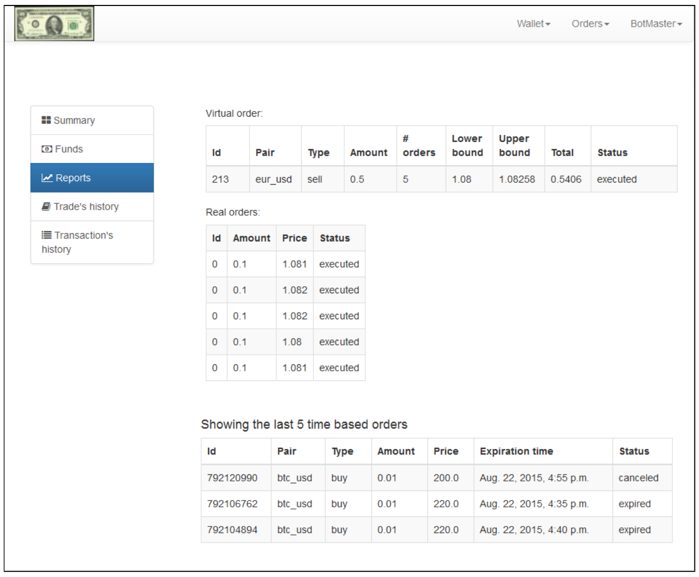
Virtual order:

Id	Pair	Type	Amount	# orders	Lower bound	Upper bound	Total	Status
209	btc_usd	buy	0.03	3	228.0	228.3	6.843	executed

Real orders:

Id	Amount	Price	Status
791923265	0.01	228.0	executed
0	0.01	228.1	executed
0	0.01	228.2	executed

Figure 7.8: Reports page 1.



Virtual order:

Id	Pair	Type	Amount	# orders	Lower bound	Upper bound	Total	Status
213	eur_usd	sell	0.5	5	1.08	1.08258	0.5406	executed

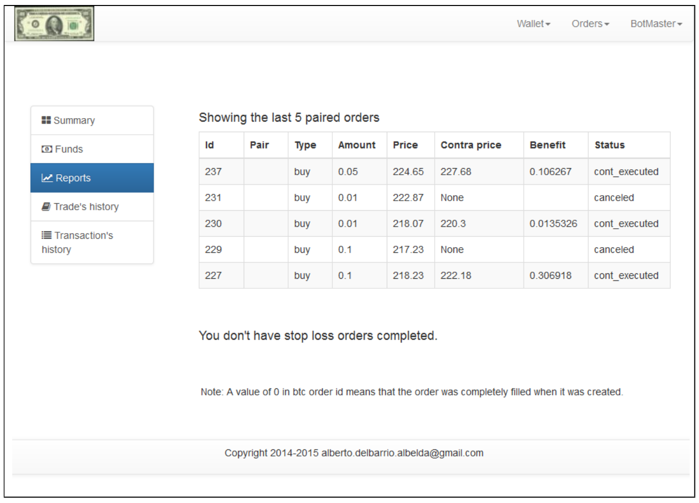
Real orders:

Id	Amount	Price	Status
0	0.1	1.081	executed
0	0.1	1.082	executed
0	0.1	1.082	executed
0	0.1	1.08	executed
0	0.1	1.081	executed

Showing the last 5 time based orders

Id	Pair	Type	Amount	Price	Expiration time	Status
792120990	btc_usd	buy	0.01	200.0	Aug. 22, 2015, 4:55 p.m.	canceled
792106762	btc_usd	buy	0.01	220.0	Aug. 22, 2015, 4:35 p.m.	expired
792104894	btc_usd	buy	0.01	220.0	Aug. 22, 2015, 4:40 p.m.	expired

Figure 7.9: Reports page 2.



Showing the last 5 paired orders

Id	Pair	Type	Amount	Price	Contra price	Benefit	Status
237		buy	0.05	224.65	227.68	0.106267	cont_executed
231		buy	0.01	222.87	None		canceled
230		buy	0.01	218.07	220.3	0.0135326	cont_executed
229		buy	0.1	217.23	None		canceled
227		buy	0.1	218.23	222.18	0.306918	cont_executed

You don't have stop loss orders completed.

Note: A value of 0 in btc order id means that the order was completely filled when it was created.

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.10: Reports page 3.

7.3.5 Trade history

The trade history section shows historical data provided by btc-e regarding the orders of a user, just mapping the data returned by the *trade_history()* API method. So, all the data displayed here is obtained in real time via btc-e, and it is not stored inside the database of the application. The view gives to the user the capacity of make queries to the API filtering and selecting the amount of received information via a form in the page. This data is

displayed inside a table.
As other parts of the btc-e API this is not documented at all, is difficult to understand the significance of the data provided and seems to be more like information of how btc-e works internally. A very good study of the information given by the trade history method and by the transaction history method has been realized by someone in internet who tried to explain the significance using two examples of operation done and data collected[3]. The summary of the research is that trade history records the amount and rate of the order while transaction history records the sub orders in which one order if filled.
The next section explains how a user can make a query with different parameters, how this query is executed and how the results are displayed to the user.

Form

The form is used by a user for select parameters when querying to get his history of trades. There are eight different parameters, and one field in the form for each one. In the lines below are described all the fields.

- pair: Represents a valid btc-e pair.
- nfrom: Get the recent orders since this order id.
- count: Maximum number of trades to get.
- from_id: Id of order to start the query.
- end_id: Id of order to finish the query.
- sort: For sorting by alphabet the result of the query.
- since: Date and time to start the query from.
- end: Date and time to end the query from.

View

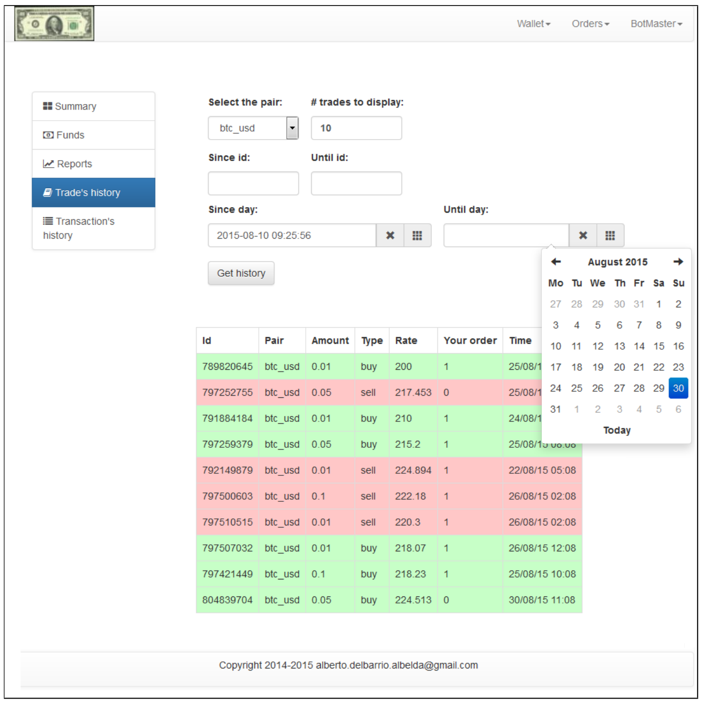
The trade history view is very simple. When the method requested by the user is an HTTP GET, the function renders the form. However if the request method is an HTTP POST, it checks if the data introduced in the form is clean, if it is, it queries the btc-e API with using the *trade_history()* method. The result of the query is stored in a dictionary and sent to the template for displaying it.

[3]`https://bitcointalk.org/index.php?topic=361052.0`

Template

The template shows the form and the results of the query if available. The information displayed by the table with the results includes: id, pair, amount, rate, type, pair and is_your_order. Like with other parts of the API, the last field is not documented and there are only people guessing about its meaning. The supposition with more sense says that the field has a value of false when their system does not fill the orders right away and their system generates new orders for smaller amounts to split up the original order into multiple new orders. This smaller orders are marked as $is\_your\_order = 0$.

Screenshot



Select the pair: btc_usd

trades to display: 10

Since id:

Until id:

Since day: 2015-08-10 09:25:56

Until day:

Id	Pair	Amount	Type	Rate	Your order	Time
789820645	btc_usd	0.01	buy	200	1	25/08/1
797252755	btc_usd	0.05	sell	217.453	0	25/08/1
791884184	btc_usd	0.01	buy	210	1	24/08/1
797259379	btc_usd	0.05	buy	215.2	1	25/08/15 08:08
792149879	btc_usd	0.01	sell	224.894	1	22/08/15 05:08
797500603	btc_usd	0.1	sell	222.18	1	26/08/15 02:08
797510515	btc_usd	0.01	sell	220.3	1	26/08/15 02:08
797507032	btc_usd	0.01	buy	218.07	1	26/08/15 12:08
797421449	btc_usd	0.1	buy	218.23	1	25/08/15 10:08
804839704	btc_usd	0.05	buy	224.513	0	30/08/15 11:08

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.11: Trade history page.

7.3.6 Transaction history

Transaction history is where a user can query the btc-e API to get a list of the lasts transactions. Looking into the information provided by the query a user can see how much money is paying in concept of transaction fee, how much money was hold by btc-e as a credit to ensure the correct payment of their fee,

the result of an order including the time when it was filled or canceled and information regarding withdrawals of money to external accounts or ingress of money to the platform.
The section in developed similarly to the trade history: there is a form where a user can refine its search to get the results he is looking for, and the results of the query are displayed in a table just under the form.
This information is taken in real time from the btc-e API.

Form

The form has seven fields to filter the received information:

- nfrom: Get the recent orders since this order id.
- count: Maximum number of trades to get.
- from_id: Id of order to start the query.
- end_id: Id of order to finish the query.
- sort: For sorting by alphabet the result of the query.
- since: Date and time to start the query from.
- end: Date and time to end the query from.

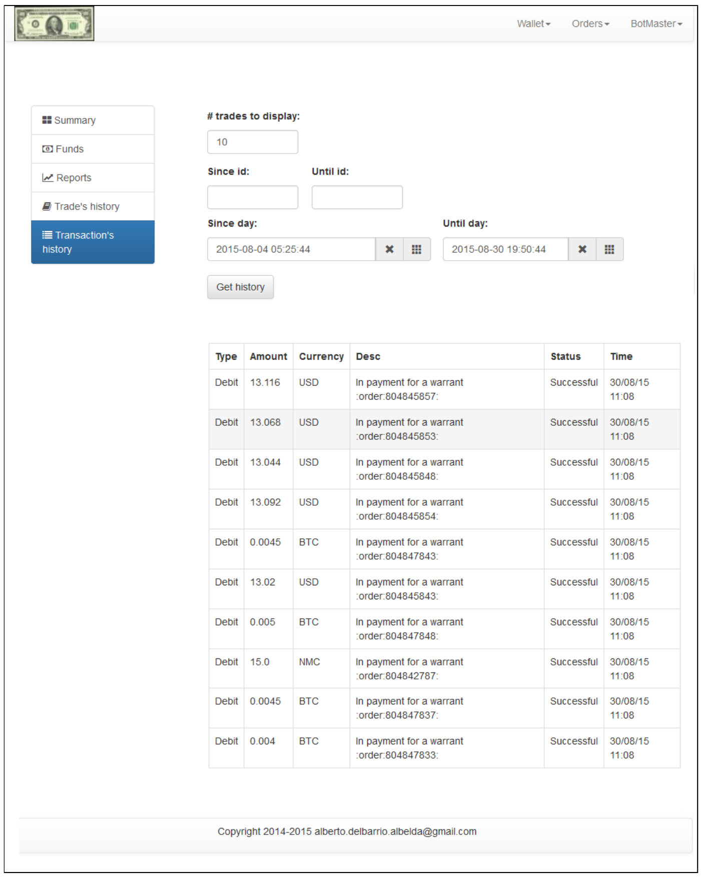
View

The view of transactions history is in charge of validate the information introduced by the user using the form, make the query to the btc-e API, transform the response and pass to the template to be rendered.
The transformation of the response given by the API is simply a translation of the status code field. Instead of showing to the final user an integer, which will not make sense for him, this code is changed for the description. There are a total of eight status codes, someone putting attention in the code below will notice that are not sorted in the *if/else* block, this is because the most common status code is the number five, for this reason its occupying the first statement, trying to make the code as optimal as possible.
This translation has been improved once btc-e put the explanation of each status code in the API "documentation", at the beginning the translation of the status codes was done guessing about the meaning of each one.

Template

The template in charge of rendering the form and the result of the query is straightforward. It displays the form inside a table without borders to get all the elements aligned and overriding some CSS styles for getting the desired visual result.
After, it checks if has a dictionary with the results of a query. If yes, it creates a table and a header row, and loops over the data creating one row per each transaction.

Screenshot



# trades to display:	10
Since id:	
Until id:	
Since day:	2015-08-04 05:25:44
Until day:	2015-08-30 19:50:44

Type	Amount	Currency	Desc	Status	Time
Debit	13.116	USD	In payment for a warrant :order:804845857:	Successful	30/08/15 11:08
Debit	13.068	USD	In payment for a warrant :order:804845853:	Successful	30/08/15 11:08
Debit	13.044	USD	In payment for a warrant :order:804845848:	Successful	30/08/15 11:08
Debit	13.092	USD	In payment for a warrant :order:804845854:	Successful	30/08/15 11:08
Debit	0.0045	BTC	In payment for a warrant :order:804847843:	Successful	30/08/15 11:08
Debit	13.02	USD	In payment for a warrant :order:804845843:	Successful	30/08/15 11:08
Debit	0.005	BTC	In payment for a warrant :order:804847848:	Successful	30/08/15 11:08
Debit	15.0	NMC	In payment for a warrant :order:804842787:	Successful	30/08/15 11:08
Debit	0.0045	BTC	In payment for a warrant :order:804847837:	Successful	30/08/15 11:08
Debit	0.004	BTC	In payment for a warrant :order:804847833:	Successful	30/08/15 11:08

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.12: Transaction history page.

7.4 Orders

The orders application is where a user of the platform can create orders against the btc-e exchange market. In has several kind of orders for make easy the work or a trader. For example using the orders application a user can create an order which will expire at certain time, or can create several orders between a defined boundaries.

7.4.1 Active orders

Active orders is the place where a user can see all the current open orders. The idea behind this section is to provide a centralized place where is easy and fast to have an overview of the different open orders of a user.
Taking advantage of the display of these orders, this is also the place where a order can be canceled. This goal is achieved using the maximum simplicity and cleanliness possible regarding the way in which is displayed and the code which allows this functionality.
For this reason not all the information that could be fetched and processed from each kind of order is displayed here, leaving this task to the reports section 7.3.4, where all the information is gathered, processed and displayed to the user once the order is finished.

View

The active orders view allows a user to see and cancel active orders. These two functionalities are separated in different functions in the views file.
The *activeOrders* function fetches the information needed for displaying the orders to the user. For each type of order it queries the database to obtain a list with the orders which belong to the current user and have as status "started". After it adds all the fields which are important for displaying into the template to a context dictionary, finally returning the rendered HTML. This function is beautiful to see because is using the same length for the dictionary variables making it easy to read.

Listing 7.2: Active orders view.

```
@login_required
def activeOrders(request):
    '''
    Render a page with a user's active orders.

    Each order can be canceled from this page.
    Each order shows its different attributes agrupted by column.
    '''
    simple =
        SimpleOrder.objects.filter(user=request.user).filter(status =
            'started')
    sliced = SlicedOrder.objects.filter(user =
        request.user).filter(status =
            'started')
    timed = TimeBasedOrder.objects.filter(user =
        request.user).filter(status =
            'started')
    paired = PairedOrder.objects.filter(user =
        request.user).filter(~Q(status =
            'canceled') & ~Q(status = 'cont_executed'))
    stoploss = StopLossOrder.objects.filter(user =
        request.user).filter(status =
            'started')

    simp = {}
    slic = {}
    pair = {}
    time = {}
    stop = []
    for o in simple:
        id = str(o.btceid)
        simp[id] = {}
        simp[id]['pair'] = o.pair.label
        simp[id]['amount'] = o.amount
        simp[id]['type'] = o.buysell
        simp[id]['price'] = o.price
        simp[id]['total'] = o.amount * o.price
    for o in sliced:
        id = str(o.id)
        slic[id] = {}
        slic[id]['pair'] = o.pair.label
        slic[id]['amount'] = o.amount
        slic[id]['type'] = o.buysell
        slic[id]['number'] = o.numberOfOrders
        slic[id]['upper'] = o.upperBound
        slic[id]['lower'] = o.lowerBound
        slic[id]['total'] = o.totalAmount()
        slic[id]['active'] = o.numActive()
        slic[id]['executed'] = o.numExecuted()
    for o in paired:
        id = str(o.id)
        pair[id] = {}
        pair[id]['pair'] = o.pair.label
        pair[id]['amount'] = o.amount
        pair[id]['type'] = o.buysell
        pair[id]['price'] = o.price
```

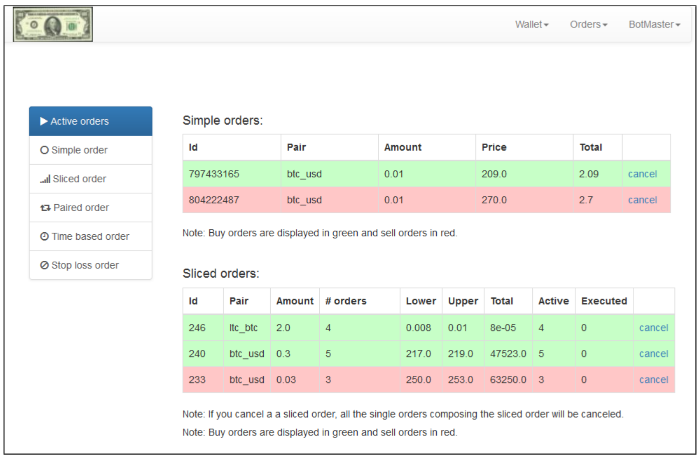
```
            pair[id]['total'] = o.amount * o.price
            pair[id]['cprice'] = o.contraprice
        for o in timed:
            id = str(o.id)
            time[id] = {}
            time[id]['pair'] = o.pair.label
            time[id]['amount'] = o.amount
            time[id]['type'] = o.buysell
            time[id]['price'] = o.price
            time[id]['total'] = o.amount * o.price
            time[id]['exp'] = o.expiration_time
        for o in stoploss:
            id = str(o.btceid)
            stop[id] = {}
            stop[id]['pair'] = o.pair.label
            stop[id]['amount'] = o.amount
            stop[id]['type'] = o.buysell
            stop[id]['price'] = o.price
            stop[id]['total'] = o.amount * o.price

        context={'simple': simp, 'sliced' : slic , 'paired': pair,
            'time': time}
        return render(request,'orders/activeorders.html',context)
```

As was mentioned before, each order shown here must be cancelable just doing one click. To achieve this goal, one view has to be created to cancel each type of order, because the code in charge of due the actual cancellation depends on the kind of order, even if they look pretty similar.

Template

This template will create as much tables as types of active orders a user has. If the user does not have active order of one kind, it will show a message saying it.

Screenshot



Simple orders:

Id	Pair	Amount	Price	Total	
797433165	btc_usd	0.01	209.0	2.09	cancel
804222487	btc_usd	0.01	270.0	2.7	cancel

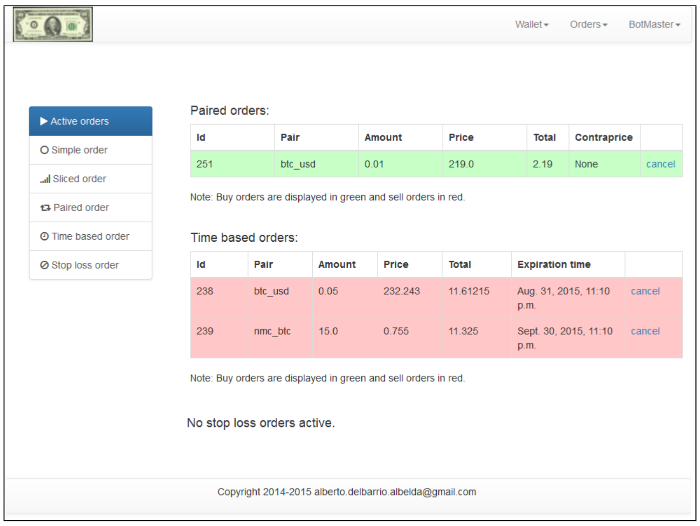
Note: Buy orders are displayed in green and sell orders in red.

Sliced orders:

Id	Pair	Amount	# orders	Lower	Upper	Total	Active	Executed	
246	ltc_btc	2.0	4	0.008	0.01	8e-05	4	0	cancel
240	btc_usd	0.3	5	217.0	219.0	47523.0	5	0	cancel
233	btc_usd	0.03	3	250.0	253.0	63250.0	3	0	cancel

Note: If you cancel a a sliced order, all the single orders composing the sliced order will be canceled.
Note: Buy orders are displayed in green and sell orders in red.

Figure 7.13: Active orders page 1.



Paired orders:

Id	Pair	Amount	Price	Total	Contraprice	
251	btc_usd	0.01	219.0	2.19	None	cancel

Note: Buy orders are displayed in green and sell orders in red.

Time based orders:

Id	Pair	Amount	Price	Total	Expiration time	
238	btc_usd	0.05	232.243	11.61215	Aug. 31, 2015, 11:10 p.m.	cancel
239	nmc_btc	15.0	0.755	11.325	Sept. 30, 2015, 11:10 p.m.	cancel

Note: Buy orders are displayed in green and sell orders in red.

No stop loss orders active.

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.14: Active orders page 2.

7.4.2 Base order

Model

The base order represents the simplest kind of order in the application. All other kinds of others inherit from base order.
The first approach for building the base order model definition was to use

a Python abstract base class[4]. Abstract base classes are a special form of interface which check its sub classes ensuring that they implement particular methods. By defining an abstract base class, someone can define a common API for a set of subclasses. This capability enforce subclasses to implement the base methods, if this methods are not implemented, an error will be raised when defining the class. A similar behavior can be achieve using functions as *hasattr()*, but the error if not implemented will raise when using the method in stead of when defining the class, leading to a later detection of the bug. Many Python core modules are using ABCs like the collections module.
However the use of abstract base classes was not recommended in early versions of Django, even if can be achieved using the attribute "abstract = True" in the meta data definition inside the class. For simplicity, in stead of this approach a more traditional way of achieve the same functionality has been used, raising an error when a subclass of base order want to use a method not implemented.
The four methods which a subclass must implement are: *create*, *execute*, *cancel* and __*str*__.

- create: This method is the actual constructor of the class. The first idea was to override the __*init*__*()* method, but this is not a good idea in because Django expects the signature of a model's constructor to be in the form of *(self, *args, **kwargs)*, and from the official documentation encourage to not do it[5]. For this method to be able to act as a constructor it has the decorator @classmethod, receiving a class as first argument[6].

- execute: It is used to put an order into the market, using different kind of technique depending on the order. This method uses the decorator @property to achieve a similar role than the traditional getters and setters[7].

- cancel: This method will remove a order from the market and change its status to 'canceled'. It also uses the @property decorator.

- __*str*__: This method overrides the representation of the class as a string to make it more readable when debugging.

[4]`https://docs.python.org/2/library/abc.html`
[5]`https://docs.djangoproject.com/en/1.7/ref/models/instances/#creating-objects`
[6]`https://docs.python.org/2/library/functions.html#classmethod`
[7]`https://tinyurl.com/poxzt2f`

The attributes defined by base class are the minimum common between the different classes: timestamp, user and status.

- timestamp: autogenerated UNIX timestamp. It will get filled in the moment of saving the order into the database.
- user: field containing a foreign key to the user 7.2 owner of the order.
- status: string with a valid status for an order. This status will depend of the kind of order.

Template

The order base template defines a structure for controlling how all the orders are represented in an HTML document, extending the base template **??**. This decided structure uses the main grid twelve, defining in only one row four sections of same size.
The first section is reserved for displaying the secondary navigation menu, the second one is designed to contain a table with the current funds of a user, the third section is where the form of each order is presented and the fourth section can include help instructions to explain how the order works.
The secondary navigation menu contains the same items than the order's drop down menu, but makes easier and faster for a user to move around the order application. The items which appear are: active orders, simple order, sliced order, paired order, time based order and stop loss order.
The menu is wrapped inside the correspondent *grid_3 class*, and inside it has two more sections wrapped into divs. The first one is "mini-submenu" class, which only has "span" sections containing nothing but the definition. The purpose of these lines is not so easy to find, because it is not well documented in the Bootstrap documentation but there is a StackOverflow question[8] which explains it: they are use for rendering a nice menu when the screen is small, making the HTML responsible. The next div is a "list-group" class which contain a list of items, one per sub order section. Each of the sub elements are "list-group-item" class and can be of other special class: "active". As can be observed in the code below, there is an if condition inside every item, which will add the class "active" only if the browsed URL contains the link to which is pointing the item. If an item is of "active" class its background color will change, making easy to distinguish the current section. This goal is a common component that many websites developed using variety of framework are looking for. Normally this is achieved using

[8] `https://tinyurl.com/pkswn7a`

Javascript for changing the class, for that the user has to execute a little piece of code in the browser. However the approach followed in this document has been to use Django to add the class in order to use as less Javascript as possible.
The funds block displays the current funds of a user. In the beginning of the development, this space was defined but the HTML that generate the table with the funds was placed in every order. In the second iteration of the web UI development that was changed, and the funds are displayed now in the base template. This decision has been taken to make the web UI easier to maintain following the DRY[9] principle. The table is displayed using three classes from Bootstrap: *table* (main class to render a table), *table-hover* (adding the hover effect on every row) and *table-bordered* (surround the table with borders). The table is composed by the header and the body. The header contains the title of two of the three column: currency and amount. The first column will display the icon of each currency, and the other two rows the name and the amount of the currency that a user has currently available. In the case that the user has no available funds, a message indicating that there are not available funds will be displayed in stead of the table.
Then, appears the order block which is empty, because the actual form to render depends of the kind of order.
The last block is the help block, empty as well for the same reason. This block will contain a little help message or instructions to make easier to understand how to set the order.

7.4.3 Btce order

Btce orders are the minimum valid orders to interact with btc-e.com. It matches the attributes that a trader will set if using directly the exchange market to place an order.

Models

The btce order model is inheriting from the base order model and overriding all its methods. Here there is an explanation of each one.

- create: Initialize a Btce order for a user.
- execute: Calls the create_order method **??** of the btc-e.com API to add an order into the market. It will return error in case that the result of the execution was not positive. The error message is the same that

[9] `https://en.wikipedia.org/wiki/Don%27t_repeat_yourself`

returns the API. If there is no error, the status attribute will change to "started" saving the returned order id into the database.

- cancel: Calls the cancel_order method 8.5.4 method of the btc-e.com API to cancel an order. It will be successful if the id of the order exists for a certain user, changing the status in the database to "canceled". It will return error in other case.

It also adds 5 attributes to the parent class:

- *buysell*: String which says if an order is for selling or for buying.
- *pair*: Foreign key to a valid btc-e pair.
- *amount*: Quantity of coins to exchange.
- *status*: String representing all the possible states where a Btce Order can be. These are: created, started, canceled and executed.
- *btceid*: Id returned by btc-e.com in a valid order creation. This identifier is used internally by the exchange market.

7.4.4 Simple order

A simple order represents a transaction in btc-e.com, in terms of database model it looks like its parent class: Btce order. The meaning of having a simple order is to be able to distinguish between an order placed by a user using the correspondent view in the web application, and an order which is part of a more complex order, like a suborder in a sliced order.
At the beginning of the development, this class was not existing leading to problems in the code in situations like: when a user creates a Simple order via the web interface, a instance of this model is stored in the database. When a user creates a sliced order, many simple orders will be created and added it to the database. In this situation there is no way to distinguish between the Simple orders created by the user using the correspondent view and the one created by the application for the sliced order. So the generation of the reports was difficult and there was no possibility of list all the active simple orders for a user.

Model

As explained before, the Simple order model is exactly the same as the Btce order model.

Form

The simple order form is the most basic form. It contains 4 attributes: *pair*, *amount*, *price* and *buysell*. All of these attributes are mandatory, and the user will receive an error message if he does not fill them or if inserts non allowed values.
Pair is a ModelChoiceField class which will be rendered using a Select widget. To get the list of the available options it queries the database to get all the available pairs. Can be notice that it includes a dictionary of attributes, that will be added to the HTML when the widget will be rendered. The style attribute is used to set the dimension and the class attribute is used to get a nice select element from the bootstrap.
Amount and price are *FloatFields*. As in the before parameter, it also specifies the size and the class of the element for HTML.
Buysell is a *ChoiceField* which will be rendered in HTML using a radio select widget with two options: buy and sell.

View

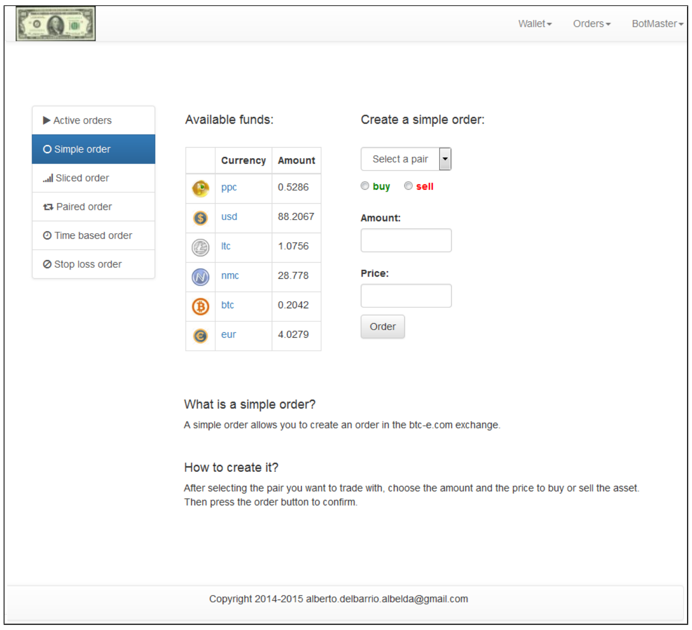
The simple order view function is the code responsible for collecting and processing the dynamic part of the simple order page.
First it gathers the available funds for a user, discarding the ones equal to zero and rounding the others to 4 decimal digits, then it gets an instance of a *simpleOrderForm* and if the HTTP method was POST, it will get the result of the simple order creation from the btc-e API.
In this moment the code looks simple, just 21 lines (without comments) but was not always like this. This function had many iterations, in the beginning the code was trying to achieve the maximum performance, what was increasing the logic and making it less readable.
For example when a user creates an order, the program queries the btc-e API for adding that order into the market, the returned JSON response includes a field with the available funds of the user after making the transaction. In the firsts versions the funds returned where used, making not necessary to query again the API to get the funds. With this approach the code had a bunch of lines more, making it more complex for reading and understanding. However now the funds returned are discarded and an explicit call to the API for having the funds is done. Is easy to understand that this approach is less efficient and, in some cases, more time consuming due to the increase number of calls to an external API. Consequently the less optimal code was elected trying to benefit readability in favor of optimality. The use of the returned funds is just an example, as it can see in the history of the code

via the git repository, this function had had 32 lines, making it optimal but ugly. This approach is followed by the next views: elegance.
After the function header appears the definition of the context dictionary. This dictionary will contain all the data that will be passed to the template, also known as the dynamic data.
Later there is a condition distinguishing a POST call with a filled form and a GET call without information. In case of POST, it will make sure that all the fields are valid, adding an error to the dictionary in case some of the fields were not valid, via the built-in Django form.errors attribute. In case of valid fields, it will create a *simpleOrder* object with the correspondent data, and call its execute method. The response of the exchange will contain an error or a success message which will be added to the context dictionary.
Before returning, it will gather the user available funds and a instance of the *simpleOrder* form adding them to the context. To finish it renders the correspondent HTML template passing the context filled in the lines before.

Template

The template to render the simple order page, like all the other order templates, extends from the order base.
The first lines of code are responsible to render a possible return code of an order creation using the font color green to display a success message or red font color to display an error like "Not enough funds to make the transaction".
Continuing through the code, appears the order block, where the form to create a new simple order is rendered. Each element of the form is wrapped inside *<p>* and *<div>* elements for having a clear separation between them. The first element in the form is a list which shows all the possible pairs to trade with, after appear two radio button for decide if buy or sell, displayed with font colors green and red respectively. After there are two fields, one for the amount and another for the price of the asset. The last element in the form is a button which will send the introduced data to the system.
The last block in the code is the help block, where a little explanation about how to create a simple order is displayed.
The code described above can be found in the appendix 10.3.5.

Screenshot



Available funds:

Currency	Amount
ppc	0.5286
usd	88.2067
ltc	1.0756
nmc	28.778
btc	0.2042
eur	4.0279

Create a simple order:

Amount:

Price:

What is a simple order?

A simple order allows you to create an order in the btc-e.com exchange.

How to create it?

After selecting the pair you want to trade with, choose the amount and the price to buy or sell the asset. Then press the order button to confirm.

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.15: Simple order page.

7.4.5 Sliced order

A (Volume) sliced order is a complex type of order which slices a simple order into smaller orders based on the trading volume.
For create one, a user will set the boundaries of the price, the amount of orders that wants to create, pair, the quantity to trade and type of order. The application will create as many orders as specified by the user, with the price spread it between the two boundaries.
The sliced order strategy could be very complex depending of the criteria regarding the subdivision of the orders but in this implementation only one strategy to slice is available: equality. So the price of each order increases using a fixed amount. The next pseudo code will explain the procedure.

Listing 7.3: Pseudocode of slicing algorithm.

```
upperBound=199
lowerBound=195
numberOfOrders=20
amountToBuy=2
orderAmount= amountToBuy / 20
step = (upperBound - lowerBound) / numberOfOrders
for price=loweBoud; price <=uperBound; price=price+step
```

```
createOrder(orderAmount, price, btc/usd)
```

The use of this kind of orders is very common, mainly in two situations. When a trader wants to make a large order hided in smaller ones wishing that others market players will not see a big order. This means in technical terms to have less impact in the market.
In the other hand, it can be used when a trader wants to buy some stock without paying a maximum price. For example: supposing that the price of btc/usd is going down and currently 200. An experienced trader wants to buy 2 BTC, but is not sure if the price will continue decreasing. So he will create a sliced order with upper bound to 199, lower bound to 195 and number of orders 20. Doing this the program will generate 20 orders, with buy price starting at 195 and increasing in steps of 0.2 until 199. Note that doing this, is not guaranteed that all the orders will be executed, and could be that only some of theses orders will be executed if the price does not achieve the lowest value.

Model

The Sliced order model inherits from Base order, adding 3 more method and 7 attributes.
The attributes are:

- *simpleOrder*: foreign keys to simple orders. It will contain as many foreign keys as number of orders.
- *upperBound*: float containing the maximum price of an order.
- *lowerBound*: float containing the minimum price of an order.
- *numberOfOrders*: integer containing the number of total orders in an sliced order.
- *status*: string with the state of the order. It can be: created, started, canceled, executed.

The methods contained in the class are:

- *create(...)*: initializes the variables and saves the object into the database.
- *execute(self)*: follows the pseudo code explained before, creating orders based on a fixed step pre calculated. It saves each simple order and the sliced oder to add the new items to the database.

- *cancel(self)*: iterates over all the simple orders, if the status is "started" cancels it.
- *numActive(self)*: this helper method returns the amount of active orders inside the sliced order.
- *numExecuted(self)*: returns the number of simple orders executed.
- *totalAmount(self)*: returns the total amount of currency in an order.

Form

The sliced order form contains six attributes: pair, amount, *numberOfOrders*, *lowerBound*, *upperBound* and *buysell*. All of these attributes are mandatory, and the user will receive an error message if he does not fill them or if inserts non allowed values.
Pair is a ModelChoiceField class which will be rendered using a Select widget. To get the list of the available options it queries the database to get all the pairs. Can be notice that it includes a dictionary of attributes, this attributes will be added to the HTML when the widget will be rendered, style is used to set the dimension and class is used to get a nice select element from the bootstrap.
Amount is a FloatField. As pair did, it also specify the size and class of the element for HTML.
numberOfOrders is a IntegerField, it also add the settings that will be needed when render the HTML to get the expected design of the widget.
lowerBound is a FloatField where the user will set the price of the lowest order.
upperBound is a FloatField where the user will set the price of the more expensive order.
Buysell is a ChoiceField which will be rendered in HTML using a radio select widget which two options: buy and sell.

View

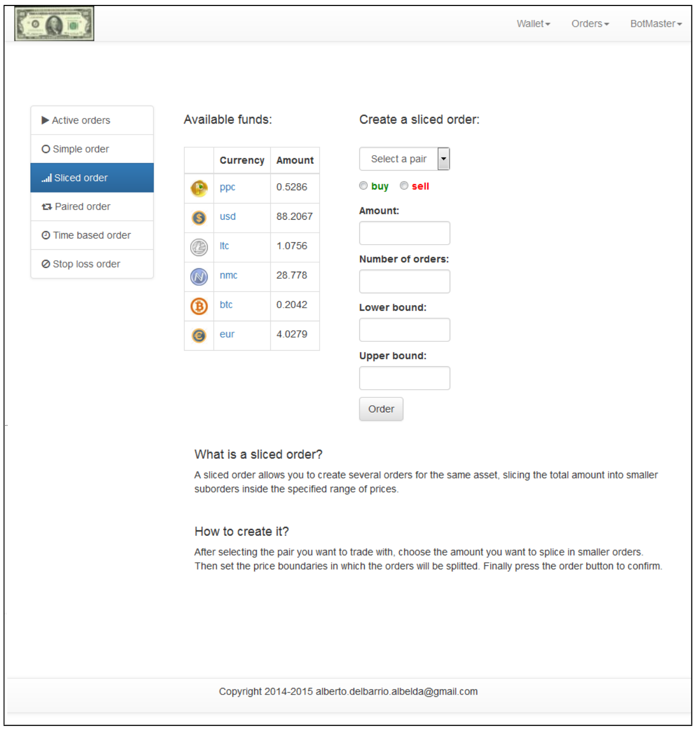
The sliced order view is the code responsible for collecting and processing the dynamic part of the *slicedOrder* page. It gathers the available funds of a user, discarding the ones equal to zero and rounding the others to 4 decimal digits, gets an instance of a *simpleOrderForm* and if the HTTP method was POST, it will get the result of the sliced order creation from the btc-e API. The code looks pretty similar to the simple order view, actually it does the

same but instancing a sliced order form instead of a simple form and a sliced order class instead of a simple.
This function has changed a lot since the first implementation, when the actual calculations to get the market price for each order were did in this view. However as the code was being refactored, this calculations were moved to the execute method of the model, simplifying a lot this function.
As mentioned above the explanation of the code structure is familiar for a reader who put attention in the simple view order, the use of the context dictionary is the same and also the validation of the form and the results obtained from the exchange API.

Template

The template which renders a sliced order web page extends the order base template adding content to the message, order and help blocks. The message has the same content as the one explained in simple order but the form is slightly different: it adds fields to specify the total amount of orders, the lower and upper bounds to set the price and it removes the price field.
The help block contains easy instructions for a user who wants to create a sliced order.

Screenshot



Available funds:

	Currency	Amount
	ppc	0.5286
	usd	88.2067
	ltc	1.0756
	nmc	28.778
	btc	0.2042
	eur	4.0279

Create a sliced order:

Amount:

Number of orders:

Lower bound:

Upper bound:

What is a sliced order?

A sliced order allows you to create several orders for the same asset, slicing the total amount into smaller suborders inside the specified range of prices.

How to create it?

After selecting the pair you want to trade with, choose the amount you want to splice in smaller orders. Then set the price boundaries in which the orders will be splitted. Finally press the order button to confirm.

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.16: Sliced order page.

7.4.6 Time based order

A time based order is a kind of limit order that allows a user to set up an expiration time for an order. This means that if before the expiration moment the order has not been executed, it will be canceled automatically. The use of time based orders becomes important in situations where the price of the market for one asset is changing fast and a user wants to create a order for the next hours, but does not want to leave it forever because is suspecting that the price will change.
This kind of orders are extensively used while trading, often the expiration time is set to the end of the day, creating like this an well known kind of order named day order[10]. However the application allows a user to set a custom expiration time increasing the flexibility of this kind of orders.

[10]http://www.investopedia.com/terms/d/dayorder.asp?version=v1

Model

The time based order is a subclass of *BtceOrder* which adds only one more attribute, expiration_time. It can be in a different status than its parent class: 'expired'.
It has the same three methods than the parent class, the purpose of them is the same but they behave different as the implementation of these orders is different.
The execute method creates the order in btc-e.com via the *btceapi.* module as the parent order does, if the order creation in btc-e has been successful, it serializes the *TimeOrder* object and sends it to the *timer_order queue*, where it will be picked up by the timer order manager 8.2.1.
The code looks pretty simple, the actual lines to send the object to the queue has been wrapped inside a function to increase the readability.

Form

The time based order form is inheriting from the parent class *simpleOrder-Form* only adding one more attribute: expiration time.
Expiration time is special because for be rendered in the HTML uses a widget not included inside the Django core. If this widget was not used, the only possibility of picking a date an time from the user was to create 5 integers fields: year, month, day, hour and minutes. This approach is not so nice and increases the complexity in the logic of validating the date.
DateTimeWidget was chosen for render the element. *DateTimeWidget* is the only widget open source available to accomplish this kind of task (notice that there are many other widgets to pick dates but normally are only Javascript, so the developer has to deal with the logic of getting the data into Django). It is based on bootstrap, so was perfect because this library is already included in the most of the HTML pages which compose the application. The widget is highly customizable, more information about it and the source code can be found in the author's page[11].

View

The *timeBasedOrder* function is the piece of code responsible for collecting and processing the dynamic part of the *timeBasedOrder* page. It gathers the available funds for a user, discarding the ones equal to zero and rounding the others to four decimal digits, gets an instance of a *timeBasedOrderForm* and if the HTTP method was POST, it will get the result of the time based

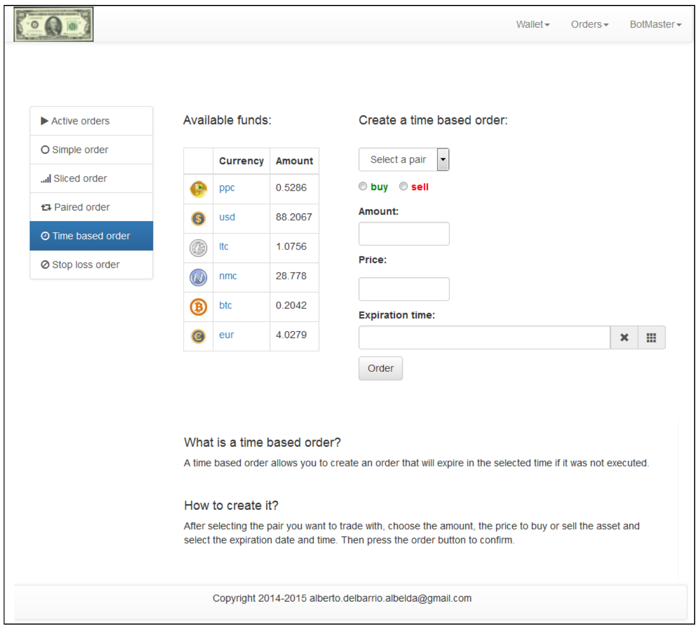
[11]`https://github.com/asaglimbeni/django-datetime-widget`

order creation from the btc-e API.
This code is also similar to the two previous views because of the coding style: hide the actual logic of the different orders behind the methods in the models. The first version of this view was ugly and messy mainly due to the uncertain idea of how to create an order which could be canceled in a certain time.
Following the style explained before, the mechanism for the cancellation is set into the backend.

Template

The time based order template renders the web page where a user can create a new order of this kind. It looks pretty similar to the other order templates, adding a widget for choosing the date and time of expiration and changing the help displayed to the user explaining how to create a new order.

Screenshot



Available funds:

	Currency	Amount
	ppc	0.5286
	usd	88.2067
	ltc	1.0756
	nmc	28.778
	btc	0.2042
	eur	4.0279

Create a time based order:

Amount:

Price:

Expiration time:

What is a time based order?

A time based order allows you to create an order that will expire in the selected time if it was not executed.

How to create it?

After selecting the pair you want to trade with, choose the amount, the price to buy or sell the asset and select the expiration date and time. Then press the order button to confirm.

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.17: Time based order page.

7.4.7 Paired order

A paired order is a composition of two simple orders trading with the same pair and same amount but going in the opposite direction one to the other.

This means that when the first (or master) order is a buying order, the second order (or slave) must be a selling order and vice versa.
A user is only allowed to create a slave order once the master order has been executed. This kind of orders are very useful when a user wants to know the exact profit obtained doing two movements.
To clarify this idea, here appears an example of a paired order.

- A user creates a master order of buying 1 BTC at 250 USD.
- Once the order is executed the user is able to create a slave order.
- The user creates a slave order of selling 1 BTC at 275 USD.
- Once the salve order ir executed, the user have won 25 USD (fees not included).

What happens when a user cancels an order? The program follows a simple policy: when a user cancels a paired order, does not matter the current phase of the order, it will be canceled and marked globally and locally as canceled. So, if the first order has not been executed, the simple order as well as the paired order will be marked as canceled; the same applies for the case when the slave order is canceled.
The logic inside the application makes sure that the four policies for a paired order are applied.

- The slave order must be trading the same pair as the master order.
- The slave order must trade the same amount as the master order.
- The slave order can be created only if the master has been executed.
- Canceling a order will mark the simple order currently active and the paired order as canceled.

The next sections explains how these constrains are achieved.

Model

The paired order model is a subclass of btce order. It adds three fields to control the slave order.

- *contraprice*: it will be null when the order is created and it will contain th price for trading with the slave order.
- *contrabuysell*: field containing the opposite action than the master order.

- *contrabtceid*: null when the paired order is created, it will have the btceid for the slave order.

The status field (inherited from btce order model) adds some more possible values to match the phases of a paired order: *cont_started* when the slave order is created, *con_canceled* if the slave order is canceled and *con_executed* when the slave order is executed.
The constructor is pretty straightforward: the first line is calculating the *contrabuysell* value using an elegant short-form from the "if" statement, also called ternary operator[12]. The next lines are initializing all the mandatory attribute of the class, saving the new object into the database and returning it.
The c*reate_contraorder* method creates and executes a simple order storing the returned id by the API of btce in its correspondent field. It has pretty the same code than simple order's *create_order* and *execute_order* methods put it together, but adding different messages for logging purposes.
The cancel method will make sure that the policy explained above regarding canceling orders is followed, logging with precision the error code. The master order uses the create and execute methods from the parent class.
As can be observed the first two policies described in the previous section (slave order have the same pair and same amount) are achieved just not creating the fields in the model for these values and taking the values from the master order, not giving possibility of error.

Form

Differently from all the other orders, the paired order is not using one but two forms for its creation: the first one allows the possibility of creating a master order, and the second one allows the possibility of creating the slave order. However this fact is transparent to the user.
The decision of split the form in two has been taken to re-use code and have simple forms.
The second form is called contra order form and it is inheriting from the base Django form. It has something special not used in any other form: a constructor. The code itself looks very simple but the implementation took some hours because the documentation regarding this scope is not very clear.
The use of the construction is needed to ensure the third policy of a paired order: a slave order can be created only if the master has been executed. To ensure it, the form is making a query to the database asking for all the paired

[12]`https://docs.python.org/2/reference/expressions.html#conditional-expressions`

orders from the current user with status ’executed’, and creating a list with the results.
Ensuring the policy could be achieved using multiple approaches but I have consider this the most elegant, other possibilities are adding this logic in the view in several ways, but this increments the complexity of the view and make it less maintainable. The code explained can be seen here:

Listing 7.4: Contra order form.

```
class contraOrderForm(forms.Form):
    '''
    This form allows a user to create the contra order of a paired
        order. Note
    that the first order have to be already executed
    '''
    def __init__(self,*args,**kwargs):
        user = kwargs.pop('user')
        super(contraOrderForm,self).__init__(*args,**kwargs)
        self.fields['opened'] = forms.ModelChoiceField(queryset =
            PairedOrder.objects.filter(user =
            user).filter(status = 'executed'), empty_label="Parent
                order", widget=forms.Select(attrs={'class':
                'form-control', 'style':'width:130px;'}),required=True)
        self.fields['contraprice'] = \
        forms.FloatField(widget=forms.TextInput(attrs =
            {'style':'width:130px;',
            'class':'form-control'}),required=True)

    opened = forms.MultipleChoiceField()
    contraprice = forms.FloatField()
```

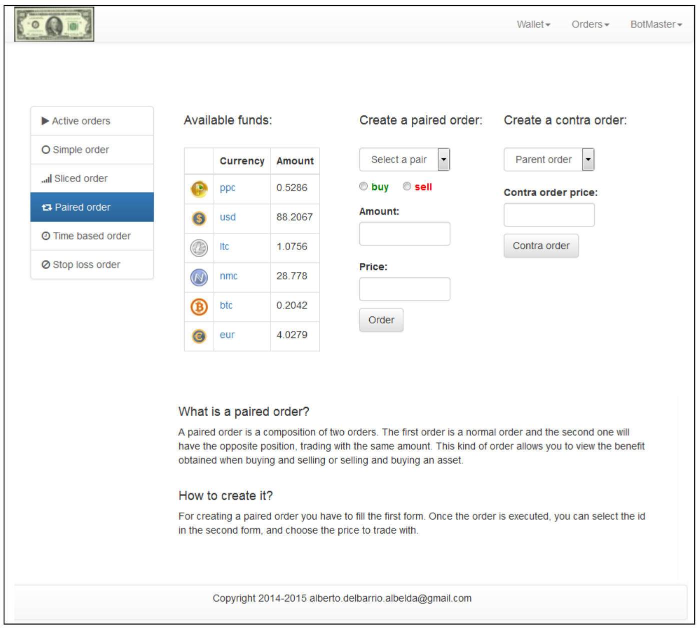
View

The code that compose the paired order view, even if a bit long, is really simple. Actually the Django best practice encourages the developer to separate any kind of logic from the view. The view gets the two forms and sends them to the template when the HTTP request is a GET. When the HTTP request is a POST, it just validates the forms adding the possible errors or success to the dictionary that will be displayed to the user. It also logs the activity using the Django logging module.

Template

The template for the paired order inherits from the orders base template. The code is easy to understand if the explanation of the previous templates has been read. It renders the two forms used by the paired order and adds some lines to help the user in the process of setting a paired order.

Screenshot



	Currency	Amount
	ppc	0.5286
	usd	88.2067
	ltc	1.0756
	nmc	28.778
	btc	0.2042
	eur	4.0279

What is a paired order?

A paired order is a composition of two orders. The first order is a normal order and the second one will have the opposite position, trading with the same amount. This kind of order allows you to view the benefit obtained when buying and selling or selling and buying an asset.

How to create it?

For creating a paired order you have to fill the first form. Once the order is executed, you can select the id in the second form, and choose the price to trade with.

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.18: Paired order page.

7.4.8 Stop loss order

A stop loss order is an order that which be canceled if the price of the market cross a defined threshold.
For example, a user creates a stop loss order for selling 10 BTC at 275 USD with a threshold of 265 USD. The current price of the pair in the market is 273 USD/BTC. If the value of the market drops to an amount minor than 265 USD/BTC, the original order will be canceled and a order to sell 10 BTC at 265 USD/BTC will be created.
With a fast reading, this kind of orders can be confused with simple orders, but they have a big difference: until the price crosses the threshold the order is not putted in place. Like this a user can have this funds available in its account.
A stop-loss order is designed to limit an investor's loss. A stop-loss order takes the emotion out of trading decisions and can be especially handy when one is on vacation or cannot watch his position. However, execution is not guaranteed in situations of a big and fast drop, but this can be achieved

enhancing the code to reflect this kind of situations. It can be known also as stop-market order[13]. This video from Investopedia provides a one minute good explanation of what is a stop order [14].

Model

The stop loss order model is a subclass of the btce order which overrides the cancel and execute method and adds two more auxiliary methods:
_send_to_queue() and *_cancel_from_queue()*.
Its class method "execute" calls the *_send_to_queue()* method to send itself to the stop loss queue where it will be picked up and managed from the stop loss backend component.
It will return success if the message was correctly introduced or error in the opposite case. Both results success and error will be logged.
Its class method "cancel" calls the *_cancel_from_queue()* method which will send a message to the queue for cancel itself.

Form

The form used to fill the information from the user to create a stop loss order is inheriting from the simple order form, not overriding any attribute or method. This because it just needs the same form fields, only changing the displaying names, where instead of "price" the user will see "Threshold" in the web page.

View

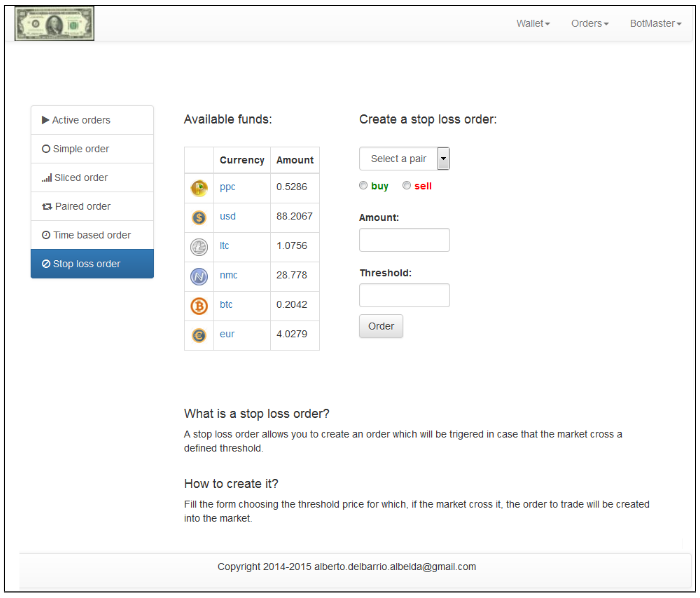
The Stop loss order view is the example of a clean and simple view. It has the minimum code to accept HTTP POST requests with the needed data from a user, creating and executing a stop loss order if the data pass the validation checks; in case data was invalid or some problem occurs during the set of the order in the market via the btc-e API, it throws an error.
It also accepts HTTP GET request for rendering the form.

Template

As can be deducted, the code composing the stop loss template looks the same than the code for the simple order form, except a different label on the price field: in the stop loss form this field is name "Threshold".

[13] `http://www.investopedia.com/terms/s/stop-lossorder.asp`
[14] `http://www.investopedia.com/terms/s/stop-lossorder.asp`

Screenshot



Available funds:

	Currency	Amount
	ppc	0.5286
	usd	88.2067
	ltc	1.0756
	nmc	28.778
	btc	0.2042
	eur	4.0279

Create a stop loss order:

Amount:

Threshold:

What is a stop loss order?

A stop loss order allows you to create an order which will be trigered in case that the market cross a defined threshold.

How to create it?

Fill the form choosing the threshold price for which, if the market cross it, the order to trade will be created into the market.

Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com

Figure 7.19: Stop loss order page.

Chapter 8

Back end

8.1 Introduction

The back end is the part of the project which interacts with the btc-e API providing the tools for handling complex orders, getting data, storing it into the database and gathering information of the market as well as of the user. It is not a standalone program but a collection of pieces designed to accomplish the task. These processes are divided in two categories: long running and scheduled processes.
The long running processes are tasks which once started will run in an infinite loop; the scheduled processes are tasks which are configured to run periodically. The long running processes are communicating with the front end and with other backend processes via queues, while the scheduled processes are writing the results of their execution to the database.

8.2 Long running processes

The long running processes are started once, entering in an infinite loop.
These processes have a main problem: if the process crashes the application will not longer be able to continue its normal behavior. The processes described below are not 100% reliables, and they may crash from time to time. This problem is solved using supervisor: a process manager that handle possible crashes and many more situations. Supervisor is highly described in section 5.4.

8.2.1 Timer order manager

The timer order manager is the process responsible to pick up and process the *TimerOrder* objects from the *timer_order_queue*.
When it is started, it waits for objects to appear into the queue. Once an object arrives, a callback is produced starting the next process: the expiration time of the order is compared with the current time. If it is minor or equal, the order is canceled via a call to its *cancel_order* method and the status of the order changes to "expired" before being saved into the database. If the expiration time did not arrived, the order is queued again waiting for its expiration time. The code explained can be found in the appendix 10.4.1.

8.2.2 Fetch tickers

The fetch tickers long runner process is in charge of getting the most recent tickers from the btc-e API and sends them to the fanout tickers queue. Many parts of the whole project will query this queue to get the value of the lasts tickers.
The approach of using a single process for getting the tickers and write them in a queue is excellent, because it reduces drastically the total amount of queries which are done against the btc-e API, reducing a lot the response time of the application. As a result, this is the only process fetching this information.
However this nice solution is in place, the project is not consistent in the use of this queue and some places are querying directly the API instead of using this queue.
The code that implement this process can be found in the correspondent section of the appendix.

8.3 Queues

Queues are an important part of the project, they perform the communication between front end and back end. The approach of using queues for communication between processes has been chosen because makes the project highly scalable. As was explained in the overview, this project uses RabbitMQ as queue framework.
The next subsections explain the tickers queue and the time order queue. These two queues are configured in a different way and they achieve different goals.

8.3.1 Tickers queue

The tickers queue is implementing the well known pattern "publisher/subscriber", where one process is writing into a queue and many publishers can subscribe dynamically to receive these messages. The queue, once initialized, has always length 1 ensuring that only the last ticker is inside it.
To create this kind of queue using RabbitMQ, an exchange with type "fanout"[1] should be created. Then, a program have to send data to the exchange, in this case tickers fetcher. From now on, the only thing remaining is to create a subscriber for this queue. Because many processes have to know the current prices of the currencies, many processes will subscribe to this queue. These processes will create a RabbitMQ queue and bind it to the exchange with the parameter "exclusive = True". This parameter ensures that when the process exits, this queue is destroyed and it frees its memory. From that moment the process will receive the data of the lasts tickets.

8.3.2 Timer order queue

The timer order queue is a worker queue which may contain time based order objects.
When a user fills the time based order form in the web interface, the information is converted into an object, serialized and written into the queue, where it will be waiting to be attended by the process manager. As opposite to the queue explained before, this queue can pile as many objects as it can store in memory. It is configured to be persistent against restarts of the main process. So, if the server which is hosting the process reboots, or the main program crashes, the information in the queue will be restore thanks to the data that RabbitMQ is writing into the disk.

8.4 Scheduled processes

All the scheduled processes are configured to run via the GNU cron daemon where each process represents a cronjob.
Cron is a time-based job scheduler in Unix-like computer operating systems. It typically automates system maintenance or administration[2] tasks. To be sure of which cronjobs are running in which interval an administrator logged in the server as the main user can run the command "crontab -l". To make

[1] urlhttps://www.rabbitmq.com/tutorials/tutorial-three-python.html
[2] `https://en.wikipedia.org/wiki/Cron`

the configuration of the cronjobs repeatable and ordered they are configured via Chef 5.2.

8.4.1 User funds

This task is in charge of getting the funds that a user has in a certain moment and writing them into the database.
It is scheduled to run 4 times at day, this granularity seems perfect for the purpose of the job: store the data in the database for future analysis. Anyway it can be configured to run with a different schedule based on the user preferences.
This job is not a standalone script but a Django management command. A Django management command is a extension of the possible actions performed by the manage.py file. The official Django documentation describes these kinds of commands as: "Custom management commands are especially useful for running standalone scripts or for scripts that are periodically executed from the UNIX crontab or from Windows scheduled tasks control panel"[3].
They have a special syntax and structure. In this case, the task is registered in the system as *fetch_user_funds* which is the name of the file where the code is located. It receives as a parameter, from the command line, the name of the users from which fetches the funds. The process first queries the database to get the user object extracting the btc-e API keys. After, it makes the query to btc-e to get the funds and saves the result into a dictionary. It also converts this result to a USD, saving these two dictionaries into the database. Finally it exists.

8.4.2 Feed executed orders

The feed executed orders process has been designed to mark the orders, which appears in the database with status "created", to "executed" if they have been executed by btc-e.
To accomplish this task, the process is following a simple approach: it queries the system database to obtain all the orders marked as active from every user in the system. After, it calls the API of btc-e to get the active orders for every user with active orders in the database. Then it compares the btc-e order ID parameter between the two lists, extracting the ones which are appearing in the database and not in the response from btc-e.
These orders have been executed, so the process will change their state to

[3]`https://docs.djangoproject.com/en/1.6/howto/custom-management-commands/`

"executed" and save them into the database. The job is running each minute, this granularity can be not enough for impatient traders but it can not run more frequently due to the Cron limitation of running with 1 minute granularity[4]. Probably this process will be redesigned in the future to be able to run more frequently.
In the moment of designing this job many alternatives were under discussion. The current one was chosen because of simplicity. If the btc-e API was better designed, was going to be easier to achieve this jobs just querying the database for the last executed orders and compare the id, instead of doing the opposite approach: query the API to get the active orders.
Until now only the simple/btc-e orders have been considered, but the other kind of orders also need to be marked as executed. Due to the nature of a virtual order, the same approach can not be taken because btc-e does not know nothing about the orders implemented in this project, so the ids can not be compared. To be able to mark these orders, each kind of virtual order has a foreign key to a btc-e order. Thanks to this approach, the process of marking the virtual orders as executed becomes simple: the program will loop over all the active orders and will check the status of the simple orders. If it can find an order for which all the child orders have been executed or canceled, it will mark this order as executed.
The implementation of the code can be found here 10.4.4.

8.4.3 Store tickers

This is the process responsible to fetch the tickers for each available pair of btc-e, to convert them to a valid object and to store it into the database.
This information is very important to be able to make artificial intelligence algorithms which will take decisions based in the historical price of an asset.
This scheduled job runs each 4 minutes. This granularity seems enough for the moment but could be changed in the future to run more frequently. It is implemented as a Django custom command, the same than the feed executed orders job.
The code is very simple: it queries the btc-e API to get the lasts ticker, after it creates a Fund object filling all the possible attributes, which are exactly the same than the btc-e API returns. Finally it writes each object into the database.

[4]`https://tinyurl.com/p7rdphv`

8.5 Btc-e API

8.5.1 Introduction

The API provided by btc-e.com[5] is not documented at all. It only provides the names of the methods that a user can use giving few response examples without any explanation of the fields. However it provides the information for building a rich platform on top of it.
The API is implementing REST services[6], which means that receives queries via HTTP POST and GET methods and answers to the request with JSON[7] objects.
It is divided in two parts: public and private API. The public API (version 2) could be accessed by everyone and provides methods to consult the current prices of the different currencies, the fees applied to the transactions, the lasts orders performed and the list of new orders waiting to be executed. While the private API only can be accessed by registered users. To be able to talk to the API, a registered user has to use a combination of two keys: the public and the private key. The use of the two keys is very common nowadays for implementing authentication using public key cryptography[8]. These keys can be generated using the web interface under User settings. A user can generate multiple keys and give different rights to the key. Thus a key can have no privileges (usefulness key), info privileges (someone using this key will be able to read the user data), or full privileges for being able to create and cancel orders as well as to get the data information. However withdraw money is not allowed (currently) using the API, only via web interface.
At the beginning of the project the available API was the version 2, but on January of 2015 btc-e released the new version 3, which was enhancing its public methods.
The main change between the versions is the possibility to get all the tickers in one single GET request. In version 2, only one ticker could be fetched in one HTTP GET request. This was increasing the time needed to feed all the available tickers. The new version implements a new method more: ticker's info. This method retrieves information about a ticker, such as the numbers of decimal digits allowed for a transaction and the maximum and minimum prices that can be set for a market operation.
In the next two sections, the methods supported by the btc-e API are ex-

[5]`https://btc-e.com/api/documentation`
[6]`http://en.wikipedia.org/wiki/Representational_state_transfer`
[7]urlhttp://en.wikipedia.org/wiki/JSON
[8]`https://en.wikipedia.org/wiki/Public-key_cryptography`

plained as well as an example of a response for each method. Noteworthy that if some of this methods fail, the answer is going to be almost the same, only changing the error message. Here is an example of a wrong request.

```
{"success":0,"error":"<error text>"}
```

8.5.2 Nonce generator

For authentication purposes and as a security measure the API of btc-e needs a cryptographic nonce[9] in each POST request sent by a user. Using a nonce, it ensures that no reply attacks can be done with an intercepted package.
The mechanism is simple: when a user creates a new pair of API keys, a nonce with value equal to zero is assigned to these keys. Each request done with these keys has to have a bigger value in the nonce parameter. Like this if an attacker intercepts a user's package and try to replies it, the API of btc-e will return a message saying: "Error: Invalid nonce value". The maximum nonce value accepted by API is 4294967294.
Due to this feature, the application needs a method for generating a bigger number for each request done. As a first and simple approach, the current timestamp of UNIX time was used. This has a big limitation: only one request can be done per second.
The second approach used for this generation was to store in the database the value of the last nonce used. It also has a big limitation: each request of a user will hit the database, and this can create performance problems if the system has several users trading fast.
Currently a more sophisticate method, but still simple, is used. The code for generating the nonce is reported below.

```
def get_nonce():
  t=time()
  t=t-1400000000
  t=round(t,2)
  t=int(t*100)
  return t
```

The function starts taking the current timestamp in a UNIX epoch format and it subtracts 1400000000 to the timestamp for making the number smaller. After it rounds the number to leave it with just two decimal numbers. This number is multiplied per 100. Using this approach the system is able to generate in one second 100 possible nonce values, which seems more than

[9]http://en.wikipedia.org/wiki/Cryptographic_nonce

enough even for the fastest trader.

8.5.3 Public methods

Get tickers

With this method a user can fetch the current price of a pair. To obtain it, he has to specify the pair in the GET request. This is an example of a query: "https://btc-e.com/api/2/btc_usd/ticker" And this is the answer.

```
{
  "ticker":{
    "high":219,
    "low":214.39,
    "avg":216.695,
    "vol":1276886.1837,
    "vol_cur":5894.91613,
    "last":218.093,
    "buy":218.748,
    "sell":218.093,
    "updated":1423779727,
    "server_time":1423779727
  }
}
```

Get fee

With this method, the fee applied to a transaction can be obtained. A user has to specify the pair to obtain the corresponding fee. In this example the query is for the btc_usd: "https://btc-e.com/api/2/btc_usd/fee". This is the response:

```
{"trade":0.2}
```

Get trades

Calling this method, a user can obtain the list of the lasts executed orders for a specific pair. This is an example of a query:
"https://btc-e.com/api/2/btc_usd/trades" and the response (reduced):

```
[
  {
     "date":1423779937,"price":218.048,
     "amount":0.0779,"tid":51850452,
     "price_currency":"USD","item":"BTC","trade_type":"bid"
```

```
   },
  {
     "date":1423779887,"price":218.048,
     "amount":0.153432,"tid":51850449,
     "price_currency":"USD","item":"BTC","trade_type":"bid"
   },
  {
     "date":1423779887,"price":218.048,
     "amount":0.121068,"tid":51850448,
     "price_currency":"USD","item":"BTC","trade_type":"bid"
   },
   ...
]
```

Get depth

The last method provided by the public btc-e API allows a user to know the nearest trades to the current price for a valid pair. The response is a JSON object containing asks and bids. An example of a query is: "https://btc-e.com/api/2/btc_usd/depth/" and this returns (reduced):

```
{
  "asks":[
    [218.746,0.101],[218.75,0.29027698],[218.751,0.49818532],
    [218.9,0.01],[218.903,0.013],[218.939,6.685],[218.942,1.8782],
    [218.947,0.013],[218.949,0.02],[218.95,41.23056738],
    [218.997,0.01896715],[218.998,0.021],[218.999,15.22902953],
    [219.003,0.5575556],[219.008,0.0202],[219.01,1.00474558],
    ...
  ],
  "bids":[
    [218.001,0.041],[218,84.03212548],[217.98,2.767],
    [217.363,0.01010096],[217.362,0.04],[217.301,0.202],
    [217.168,0.01109059],[217.156,0.001],[217.112,4.3592225],
    [217.083,0.01109059],[217.072,0.013],[217.071,0.011],
    [217,9.2281494],[216.986,0.026],[216.985,0.0101],
   ...
  ]
}
```

8.5.4 Private methods

The next methods are only available for a registered user.

Get information

This method is called without any parameter. The answer shows the current funds of the user, the rights of the used key, the number of transactions active, the quantity of active orders and the server time. Example of an answer:

```
{
  "success":1,
  "return":{
  "funds":{
    "usd":325,
    "btc":23.998,
    "ltc":0,
    "ruc":0,
    "nmc":0
    },
  "rights":{
    "info":1,
    "trade":1
  },
  "transaction_count":80,
  "open_orders":1,
  "server_time":1342123547
  }
}
```

Transaction history

This method returns the history of the transactions done by one user. This kind of movements can be done with both cryptocurrencies and standard currencies. It has seven optional parameters:

Parameter	Description	Values	Default
from	The ID of the transaction to start displaying with	integer	0
count	The number of transactions for displaying	integer	1,000
from_id	No The ID of the transaction to start displaying with	integer	0
end_id	The ID of the transaction to finish displaying with	integer	current
order	sorting	'ASC' or 'DESC'	DESC
since	When to start displaying	timestamp	0
end	When to finish displaying	timestamp	current

Table 8.1: Possible values for query the transaction history

The return JSON object looks like this:

```
{
  "success":1,
  "return":{
    "1081672":{
      "type":1,
      "amount":1.00000000,
      "currency":"BTC",
      "desc":"BTC Payment",
      "status":2,
      "timestamp":1342448420
    }
  }
}
```

Trade history

With this method a user can query the API to obtain a list of trades done. A trade represents whatever movement: buy or sell certain currency. This list could become really big in some months of trading with the system. The next parameters can limit the quantity of results obtained:

Parameter	Description	Values	Default
pair	the pair to show the transactions	valid pair	all pairs
from	The ID of the transaction to start displaying with	integer	0
count	The number of transactions for displaying	integer	1,000
from_id	No The ID of the transaction to start displaying with	integer	0
end_id	The ID of the transaction to finish displaying with	integer	current
order	sorting	'ASC' or 'DESC'	DESC
since	When to start displaying	timestamp	0
end	When to finish displaying	timestamp	current

Table 8.2: Possible values for query the trade history

This is a reduced result showing only one trade:

```
{
  "success":1,
  "return":{
    "166830":{
      "pair":"btc_usd",
      "type":"sell",
      "amount":1,
      "rate":1,
      "order_id":343148,
      "is_your_order":1,
      "timestamp":1342445793
    }
  }
}
```

Active orders

This method returns a JSON object containing all the current open orders of a user. This method does not accept parameters.

```
{
  "success":1,
  "return":{
    "343152":{
      "pair":"btc_usd",
      "type":"sell",
      "amount":1.00000000,
      "rate":3.00000000,
      "timestamp_created":1342448420,
      "status":0
    }
  }
}
```

Trade

Using this method a user can make a movement in the exchange market, selling or buying a certain quantity of an asset. The user must have this quantity available in his wallet. The next parameters are mandatory:

Parameter	Description	Values
pair	the pair to trade with	valid pair
type	buy or sell	'buy' or 'sell
rate	rate to buy or sell	float
amount	the amount to buy or sell	float

Table 8.3: Mandatory parameters to open an order

Here is reported a successful result.

```
{
  "success":1,
  "return":{
    "received":0.1,
    "remains":0,
    "order_id":0,
    "funds":{
      "usd":325,
      "btc":2.498,
      "ltc":0,
      "ruc":0,
      "nmc":0
    }
  }
}
```

Cancel order

Using this method a user can cancel a currently open order, specifying as a mandatory parameter the order id.

Parameter	Description	Values
orderid	id of the order to cancel	valid active order id

Table 8.4: Parameters for cancel an order

The result, if successful, looks like this:

```
{
  "success":1,
  "return":{
    "order_id":343154,
      "funds":{
      "usd":325,
      "btc":24.998,
      "ltc":0,
      "ruc":0,
      "nmc":0
    }
  }
}
```

Part III

Ending

Chapter 9

Conclusion and future work

9.1 Conclusions

Developing the platform has been a long journey, starting in June 2014 and finishing in September 2015.
During this time, I have learned a lot about Python, Django and exchange markets. I can clearly see the evolution comparing the first ugly code I was writing at the beginning with the last elegant and minimalistic code which I am writing now. This improvement can be appreciated comparing the implementation of the btc-e API module, which has remain more or less untouched since the firsts month of the development, with the code found in the orders section which is the last part I wrote.
Now I am also able to spend an entire night trading in btc-e and do not loose money, understanding which orders are well done and which ones could be improved, as well as to basically analyze the movements of other traders.
I am always trying to apply the knowledge I am acquiring working as a system administrator into the project. This knowledge can be notice in the use of a configuration management tool like Chef, or of a process manager like supervisor. I believe that combining different disciplines is the key to obtain the best results. For this reason I put my efforts into the creation of a solid system in terms of serving the application and in the proper use of the tools provided by the OS.

By the way I learned how amazing is LaTeX compared with a traditional text editor and how difficult is to write an academic paper. These are two of the things I have most enjoyed learning because in other occasion, probably I was never going to try them.

9.2 Future of the platform

My future plans for the project is to launch it into the market and maybe to win money with it.
However the project is in the very firsts steps of its development: it has to be enhanced and extended to support much more features. The most important missing components are algorithms to do auto trading, converting the current platform into a trading bot.
Also it needs to be more corporative, which means that needs a logo, a good name, a domain name and some marketing.

Before being alive, some adjusts have to be done, for example upgrading Django. Currently the project is using the version 1.6 of the framework, which was the most recent version of the software when I started, but nowadays is not longer maintained and it is deprecated in favor of the 1.8 version. The code is going to be available soon in Github, trying to catch the attention of other developers.

Chapter 10

Appendix

10.1 Configuration files

10.1.1 NGINX

Listing 10.1: Django settings.

```
error_log  /var/log/nginx/cryptomoneymakers.log;

    upstream django{
        server    unix:/var/cryptomoneymakers/nginx.sock;
    }
    server {
  listen    176.9.41.35:443 ssl;
  keepalive_timeout 70;
        ssl on;
        ssl_certificate    /etc/nginx/ssl/cert.pem;
  ssl_certificate_key /etc/nginx/ssl/cert.key;
        #To by-pass static files
        location /static {
            autoindex on;
      alias /var/cryptomoneymakers/venv/MillonesApp/static;
            #alias /static;
        }

        #Proying connections to application servers
        location / {
            include       /etc/nginx/uwsgi_params;
            uwsgi_pass    django;
            access_log /var/log/nginx/accesslog.log;
            proxy_redirect off;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For
                $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Host $server_name;
        }

    }
    server {
        listen      176.9.41.35:8080;
  location    / {
    rewrite ^ https://176.9.41.35 permanent;
        #server_name cryptomoneymakers.com;
    }
}
```

10.1.2 uWSGI

configuration file:

Listing 10.2: Django settings.

```
[uwsgi]
# Django-related settings
# the base directory (full path)
chdir           = /var/cryptomoneymakers/venv/MillonesApp/
# Django's wsgi file
module          = MillonesApp.wsgi
# the virtualenv (full path)
home            = /var/cryptomoneymakers/venv/
harakiri        = 60
# process-related settings
# master
master          = true
# maximum number of worker processes
processes       = 2
# the socket (use the full path to be safe
socket          = /var/cryptomoneymakers/nginx.sock
# ... with appropriate permissions - may be needed
chmod-socket    = 660
# clear environment on exit
vacuum          = true
uid             = cryptomoneymaker
gid             = nginx
max-request     = 200
```

10.1.3 Django

Listing 10.3: Django settings.

```python
"""
Django settings for MillonesApp project.
"""

# Build paths inside the project like this: os.path.join(BASE_DIR,
    ...)
import os
BASE_DIR = os.path.dirname(os.path.dirname(__file__))


SECRET_KEY = 'XXXXXXXXXXXXXXXXXXXXXXXXXX'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = False
TEMPLATE_DEBUG = False

ALLOWED_HOSTS = [
  '176.9.41.35',
       ]

# Application definition
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'users',
    'wallet',
    'orders',
    'south',
)

MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
)

ROOT_URLCONF = 'MillonesApp.urls'

WSGI_APPLICATION = 'MillonesApp.wsgi.application'
```

```

CSRF_COOKIE_SECURE = True

# Database
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'HOST': 'xxxxxxxxxx',
        'NAME': 'millonesApp',
        'USER': 'millonesApp',
        'PASSWORD' : 'xxxxxxxx',
        'STORAGE_ENGINE': 'MyISAM',
        }
}

# Internationalization
LANGUAGE_CODE = 'en-us'
TIME_ZONE = 'Europe/Berlin'
USE_I18N = True
USE_L10N = True
USE_TZ = True


# Static files (CSS, JavaScript, Images)
STATIC_URL = '/static/'
STATICFILES_DIR = (
    os.path.join(BASE_DIR,'static'),
    '/var/cryptomoneymakers/venv/MillonesApp/static',
    '/var/cryptomoneymakers/venv/MillonesApp/wallet/static',
)
STATICFILES_FINDERS = (
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
                  )

TEMPLATE_DIRS = (
    os.path.join(BASE_DIR,'templates'),
)
TEMPLATE_CONTEXT_PROCESSORS = (
              'django.core.context_processors.request',
              'django.contrib.auth.context_processors.auth',
              )
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format' : "[%(asctime)s] %(levelname)s
                [%(name)s:%(lineno)s] %(message)s",
            'datefmt' : "%d/%b/%Y %H:%M:%S"
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'handlers': {
        'file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename':
                '/var/cryptomoneymakers/venv/MillonesApp/cryptomoneymakers.log',
            'formatter': 'verbose'
        },
    },
    'loggers': {
        'django': {
            'handlers':['file'],
            'propagate': True,
            'level':'INFO',
        },
        'wallet': {
            'handlers': ['file'],
            'level': 'DEBUG',
        },
        'orders': {
            'handlers': ['file'],
            'level': 'DEBUG',
        },
    }
}
```

10.1.4 Chef

Default recipe

Listing 10.4: Chef default recipe.

```
#
# Cookbook Name:: cryptomoneymakers
# Recipe:: default
#
# Copyright 2015, Alberto del Barrio
#
# All rights reserved - Do Not Redistribute
#

```

```
# Install the needed packages:
%w(
python-virtualenv git python-pip gcc mariadb-devel enca librabbitmq
rabbitmq-server
).each do |p|
  package p do
    action :install
  end
end

# Create group and user
group 'cryptomoneymakers' do
  action :create
end

group 'cryptomoneymaker' do
  action :create
end

user 'cryptomoneymaker' do
  home '/var/cryptomoneymakers/'
  group 'cryptomoneymaker'
end

include_recipe 'python::default'

python_virtualenv node['crytomoneymakers']['venv_path'] do
  interpreter 'python2.7'
  owner       'cryptomoneymaker'
  group       'cryptomoneymaker'
  options     '--system-site-packages'
  action      :create
end

python_pip 'django' do
  version    '1.6'
  virtualenv node['crytomoneymakers']['venv_path']
  action     :install
end

%w(uwsgi mysql-python pycrypto Pillow iconv django-datetime-widget
pika
).each do |p|
  python_pip p do
    virtualenv node['crytomoneymakers']['venv_path']
    user 'cryptomoneymaker'
    group 'cryptomoneymaker'
    action     :install
  end
```

```
end

include_recipe 'cryptomoneymakers::server'

include_recipe 'nginx::default'
```

Server recipe

Listing 10.5: Chef server recipe.

```
#
# Cookbook Name:: cryptomoneymakers
# Recipe:: server
#
# This recipe configures the server to:
#   - serve django project with nginx
#   - roll out uwsgi config file
#   - manage the necessary cronjobs
#
# Copyright 2015, Alberto del Barrio
#
# All rights reserved - Do Not Redistribute
#

template '/etc/nginx/sites-enabled/cryptomoneymakers.conf' do
  source 'cryptomoneymakers.conf.erb'
  owner 'root'
  group 'nginx'
  mode  '0440'
  action :create
end

directory '/etc/uwsgi/' do
  owner 'root'
  group 'uwsgi'
  mode  '0770'
  action :create
end

cookbook_file '/etc/uwsgi/millones.ini' do
  source 'millones.ini'
  owner 'root'
  group 'nginx'
  mode  '0440'
  action :create
end

```

```
cron 'funds_fetcher' do
  minute '0'
  hour   '*/6'
  user   'cryptomoneymaker'
  command 'cd /var/cryptomoneymakers/venv/ &&
      /var/cryptomoneymakers/venv/bin/python
      /var/cryptomoneymakers/venv/MillonesApp/manage.py
      fetch_user_funds BotMaster'
end

cron 'tickers_fetcher' do
  minute '*/4'
  user   'cryptomoneymaker'
  command 'cd /var/cryptomoneymakers/venv/ &&
      /var/cryptomoneymakers/venv/bin/python
      /var/cryptomoneymakers/venv/MillonesApp/manage.py
      fetch_ticker_value btc_usd btc_eur ltc_btc ltc_usd ltc_eur
      nmc_btc nmc_usd nvb_btc nvc_usd eur_usd ppc_btc ppc_usd '
end

cron 'feed_executed_oders' do
  minute '*'
  user   'cryptomoneymaker'
  command 'cd /var/cryptomoneymakers/venv/ && source bin/activate &&
      /var/cryptomoneymakers/venv/MillonesApp/backend/cronjobs/feed_executed_orders.py
end


directory '/etc/nginx/ssl' do
  owner 'nginx'
  group 'nginx'
  mode  0755
end

cookbook_file '/etc/nginx/ssl/cert.pem' do
  source 'cert.pem'
  mode   '0444'
  owner  'root'
  group  'root'
end

cookbook_file '/etc/nginx/ssl/cert.key' do
  source 'cert.key'
  mode   '0400'
  owner  'root'
  group  'root'
end

# Roll out systemd unit file for uwsgi
```

```
cookbook_file '/usr/lib/systemd/system/uwsgi.service' do
  source 'uwsgi.service'
  mode   '0644'
  owner 'root'
  group 'root'
end
```

10.2 Application code

10.3 Base

Base template

Listing 10.6: Base template.

```
<!DOCTYPE html>
<html class="no-js" xmlns="http://www.w3.org/1999/html">
    <!--<![endif]-->
<html lang="en">
    <head>
        <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
        <meta charset="UTF-8">
        <!--<meta name="Cryptomoneymakers" content="Trading platform
            built on top of btc-e">
        <meta name="keywords" content="bitcoin trade btc-e">-->

        <title>{% block title %}{% endblock %}</title>
        {% load staticfiles %}
        <link href="{% static 'css/simple.css' %}" rel='stylesheet'
            type='text/css'>
        <link rel="stylesheet" href=
            "https://maxcdn.bootstrapcdn.com/
            bootstrap/3.3.5/css/bootstrap.min.css">
        <link rel="stylesheet" href=
            "https://maxcdn.bootstrapcdn.com/
            bootstrap/3.3.5/css/bootstrap-theme.min.css">
        <link rel="stylesheet" href= "//netdna.bootstrapcdn.com/
            font-awesome/4.2.0/css/font-awesome.min.css">
        <script src="https://ajax.googleapis.com/ajax/
            libs/jquery/1.11.3/jquery.min.js"> </script>
        <script src="https://maxcdn.bootstrapcdn.com/
            bootstrap/3.3.5/js/bootstrap.min.js"> </script>
        {% block header %}{% endblock %}
    </head>
    <body>
```

```
        <div class="row">
    <div class="grid_12">
            {% include 'main_nav.html' %}
          </div>
       </div><!-- end row-->

        <!-- main content area -->
        <div id="main" class="wrapper">
            {% block body_main %}{% endblock %}

        </div><!-- #end div #main .wrapper -->

  <nav class="navbar navbar-default navbar-bottom" role="navigation">
    <div class="container" style="text-align:center">
      <h5> Copyright 2014-2015 alberto.delbarrio.albelda@gmail.com
          </h5>
    </div>
  </nav>
    </body>
</html>
```

10.3.1 Main navigation menu

Listing 10.7: Navigation menu.

```
<!-- main navigation -->
  <nav id="topnav" class="navbar navbar-default" role="navigation">
    <!-- Brand and toggle get grouped for better mobile display -->
    <div class="container-fluid" style="width: 998px;">
       <div class="navbar-header">
         <button type="button" class="navbar-toggle"
             data-toggle="collapse"
             data-target="#bs-example-navbar-collapse-1">
           <span class="sr-only">Toggle navigation</span>
           <span class="icon-bar"></span>
           <span class="icon-bar"></span>
           <span class="icon-bar"></span>
         </button>
        {% load staticfiles %}
    <a class="logo"><img src="{% static 'images/logo.jpg'
        %}"text=logo"></a>
         </div>
         <div class="collapse navbar-collapse" id="navbar-collapse">
           <ul class="nav navbar-nav"></ul>
             <ul class="nav navbar-nav navbar-right">
         {% if user.is_authenticated %}
               <li class="dropdown">
```

```
              <a href="/wallet/" data-toggle="dropdown"
                  class="dropdown-toggle">Wallet<b
                  class="caret"></b></a>
              <ul class="dropdown-menu">
               <li><a href="/wallet/summary/">Summary</a></li>
               <li role="separator" class="divider"></li>
               <li><a href="/wallet/funds/">Funds</a></li>
               <li role="separator" class="divider"></li>
               <li><a href="/wallet/reports/">Reports</a></li>
               <li role="separator" class="divider"></li>
               <li><a href="/wallet/tradehistory/">Trade's
                   history</a></li>
               <li role="separator" class="divider"></li>
               <li><a
                   href="/wallet/transactionhistory/">Transaction's
                   history</a></li>
              </ul>
            </li>
            <li class="dropdown">
              <a href="/orders/" data-toggle="dropdown"
                  class="dropdown-toggle">Orders<b
                  class="caret"></b></a>
              <ul class="dropdown-menu">
                <li><a href="/orders/activeorders/">Active
                    orders</a></li>
                <li role="separator" class="divider"></li>
                <li><a href="/orders/simpleorder/">Simple
                    order</a></li>
                <li role="separator" class="divider"></li>
                <li><a href="/orders/slicedorder/">Sliced
                    order</a></li>
                <li role="separator" class="divider"></li>
                <li><a href="/orders/pairedorder/">Paired
                    order</a></li>
                <li role="separator" class="divider"></li>
                <li><a href="/orders/timebasedorder/">Time based
                    order</a></li>
                <li role="separator" class="divider"></li>
                <li><a href="/orders/stoplossorder/">Stop loss
                    order</a></li>
              </ul>
            </li>
            <li class="dropdown">
              <a href="/wallet/" data-toggle="dropdown"
                  class="dropdown-toggle"> {{ user.username }}<b
                  class="caret"></b></a>
              <ul class="dropdown-menu">
                <li><a href="/users/preferences/">Settings</a></li>
                <li role="separator" class="divider"></li>
```

```
                  <li><a href="/users/logout/">Log out</a></li>
                </ul>
                </li>
                {% endif %}
             </ul>
           </ul>
         </div><!-- /.navbar-collapse -->
       </div>
    </nav>
```

10.3.2 URLs

Listing 10.8: Base urls.

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'MillonesApp.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),

    url(r'^admin/', include(admin.site.urls)),
    url(r'^wallet/', include('wallet.urls')),
    url(r'^users/', include('users.urls')),
    url(r'^orders/', include('orders.urls')),
)
```

10.3.3 Users

Models

Listing 10.9: Users models.

```
from django.db import models
from users.models import UserProfile
from django.contrib.auth.models import User
import pickle
import pika
import logging

log = logging.getLogger(__name__)

```

```
class Currency(models.Model):
    '''
    Represents a a currency.
    '''
    label = models.CharField(max_length=3)
    name = models.CharField(max_length=20, blank = True)

    def __unicode__(self):
        return unicode(self.label)


class Change(models.Model):
    '''
    Contains the name a valid btce pair which is composed by two
        currencies.
    '''
    label = models.CharField(max_length=7)

    def __unicode__(self):
        return unicode(self.label)


class Funds(models.Model):
    '''
    This class stores the total amount of coins owned by a user in a
        time.
    currency = tav -> total available.
    currency = ttl -> total (sum available and in orders).
    '''
    currency = models.CharField(max_length=3)
    amount = models.FloatField(default=0)
    datetime = models.DateTimeField(auto_now=True,blank=True)
    user = models.ForeignKey(User)

    def __unicode__(self):
        return u'%s %s %s' % (self.currency, self.amount,
            self.datetime)

class Ticker(models.Model):
    label = models.CharField(max_length=7)
    high = models.FloatField()
    low = models.FloatField()
    avg = models.FloatField()
    vol = models.FloatField()
    vol_cur = models.FloatField()
    last = models.FloatField()
    buy = models.FloatField()
    sell = models.FloatField()
    updated = models.IntegerField()
    server_time = models.IntegerField()

    def __unicode__(self):
        return u'%s %f' % (self.label, self.average)
```

Views

Listing 10.10: Users view.

```
from django.shortcuts import render
from django.template import RequestContext
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.decorators import login_required
from django.http import HttpResponse, HttpResponseRedirect
from users.forms import *
from users.models import UserProfile
from wallet.btceapi import get_info

def user_login(request):
    '''
    This view renders the form to make possible the login of a user
    '''
    context = RequestContext(request)
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(username=username, password=password)
        if user:
            # Is the account active? It could have been disabled.
            if user.is_active:
                # If the account is valid and active, we can log the
                    user in.
                # We'll send the user back to the homepage.
                login(request, user)
                return HttpResponseRedirect('/wallet/summary/')
            else:
                # An inactive account was used - no logging in!
                return HttpResponse("Your account is disabled.")
        else:
            # Bad login details were provided. So we can't log the
                user in.
            print("Invalid login details: {0}, {1}".format(username,
                password))
            return HttpResponse("Invalid login details supplied.")

    else:
        return render(request,'users/login.html', {},)
```

```


@login_required
def user_logout(request):
    '''
    Log out a user when requests this view.
    '''
    logout(request)
    return render(request,'users/login.html',{})


@login_required
def preferences(request):
    '''
    This view shows the preferences for a user.
    '''
    return render(request,'users/preferences.html')


@login_required
def change_password(request):
    '''
    This view allows a user to change his password.
    '''
    context = {}
    if request.method == "POST":
        form = ChangePasswordForm(request.POST)
        context['cpf']=form
        if form.is_valid():
            user = authenticate(username=request.user.username,
                password=(form.cleaned_data['oldpass']))
            if user:
                user.set_password(form.cleaned_data['newpass1'])
                context['success'] = 'Password changed successfully'
            else:
                context['error'] = 'Invalid old password supplied'
        else:
            context['error'] = 'Invalid form data'
    else:
        form = ChangePasswordForm()
        context={'cpf':form}
    return render(request,'users/change_password.html', context)


@login_required
def change_api_keys(request):
    '''
    This view allows a user to change his API keys.
    '''
    context = {}
    if request.method == "POST":
        form = ChangeApiKeysForm(request.POST)
        context['akf']=form
        if form.is_valid():
            user.userprofile.btce_key = form.cleaned_data['apikey']
            user.userprofile.btce_secret_key =
                form.cleaned_data['secretkey']
            user=authenticate(username = request.user.username,
                password=(form.cleaned_data['oldpass']))
            context['success'] = 'API keys changed successfully'
        else:
            context['error'] = 'Invalid keys'
    else:
        form = ChangeApiKeysForm()
        context={'ckf':form}
    return render(request,'users/change_api_keys.html', context)


def register(request):
    '''
    A user can register using this view.
    '''
    context = RequestContext(request)
    # A boolean value for telling the template whether the
        registration was successful.
    # Set to False initially. Code changes value to True when
        registration succeeds.
    registered = False

    if request.method == 'POST':
        # Attempt to grab information from the raw form information.
        # Note that we make use of both UserForm and UserProfileForm.
        user_form = UserForm(data=request.POST)
        profile_form = UserProfileForm(data=request.POST)
        # If the two forms are valid...
        if user_form.is_valid() and profile_form.is_valid():
            user = user_form.save()
            # Now we hash the password with the set_password method.
            # Once hashed, we can update the user object.
            user.set_password(user.password)
            user.save()

            # Now sort out the UserProfile instance.
            # Since we need to set the user attribute ourselves, we
                set commit=False.
            # This delays saving the model until we're ready to avoid
                integrity problems.
            profile = profile_form.save(commit=False)
            profile.user = user
            if 'picture' in request.FILES:
                profile.picture = request.FILES['picture']
             # Now we save the UserProfile model instance.
            profile.save()
            # Update our variable to tell the template registration was successful.
            registered = True
        else:
            print(user_form.errors, profile_form.errors)

    else:
        user_form = UserForm()
        profile_form = UserProfileForm()

    return render(request, 'users/register.html',{'user_form': user_form, 'profile_form': profile_form, 'registered': registered})
```

Forms

Listing 10.11: Wallet forms.

```
from django import forms
from wallet.models import Change
from datetimewidget.widgets import DateTimeWidget


class tradeHistoryForm(forms.Form):
    cl=[]
    changes = Change.objects.all()
    [cl.append((c.label,c.label)) for c in changes]
    pair = forms.ChoiceField(required=False, widget=forms.Select(attrs =
        {'class': 'form-control', 'style':'width:130px;'}), choices=cl)
    nfrom = forms.IntegerField(widget=forms.TextInput(attrs = {'style':'width:130px;',
        'class': 'form-control', 'style':'width:130px;'}),required=False)
    count = forms.IntegerField(widget=forms.TextInput(attrs = {'style':'width:130px;',
        'class': 'form-control', 'style':'width:130px;'}), required=False)
    from_id = forms.IntegerField(widget=forms.TextInput(attrs = {'style':'width:130px;',
        'class': 'form-control',
            'style':'width:130px;'}),required=False)
    end_id = forms.IntegerField(widget=forms.TextInput(attrs =
        {'style':'width:130px;',
        'class': 'form-control',
            'style':'width:130px;'}),required=False)
    sort = forms.ChoiceField(required=False, widget=forms.RadioSelect,
            choices=(('ASC','newest first'),('DESC','oldest first')))
    since = forms.DateTimeField(required=False,
        widget=DateTimeWidget(usel10n=True,
        bootstrap_version=3, attrs={'class':'form-control',
            'style':'width:240px;'}))
    end = forms.DateTimeField(required=False,
        widget=DateTimeWidget(usel10n=True,
        options={'weekStart':1,'todayBtn':'true'},bootstrap_version=3,
            attrs={'class':'form-control'}))


class transactionHistoryForm(forms.Form):
    nfrom = forms.IntegerField(widget=forms.TextInput(attrs =
        {'style':'width:130px;',
        'class': 'form-control',
            'style':'width:130px;'}),required=False)
    count = forms.IntegerField( widget=forms.TextInput(attrs =
        {'style':'width:130px;',
        'class': 'form-control', 'style':'width:130px;'}),
            required=False)
    from_id = forms.IntegerField( widget=forms.TextInput(attrs =
        {'style':'width:130px;',
        'class': 'form-control',
            'style':'width:130px;'}),required=False)
    end_id = forms.IntegerField( widget=forms.TextInput(attrs =
        {'style':'width:130px;',
        'class': 'form-control',
            'style':'width:130px;'}),required=False)
    sort = forms.ChoiceField(required=False, widget=forms.RadioSelect,
            choices=(('ASC','newest first'),('DESC','oldest first')))
    since = forms.DateTimeField(required=False,
        widget=DateTimeWidget(usel10n=True,
        options={'weekStart':1,'todayBtn':'true'},bootstrap_version=3))
    end = forms.DateTimeField(required=False,
        widget=DateTimeWidget(usel10n=True,
        options={'weekStart':1,'todayBtn':'true'},bootstrap_version=3))
```

Base template

Listing 10.12: Users base template.

```
{% extends 'base.html' %}

{% block body_main %}
  <section id="content" style="margin-bottom:60px; width:100%"
      class="wide-content">
    <div class="row">
     <div class="grid_3">
     <div class="mini-submenu">
       <span class="icon-bar"></span>
         <span class="icon-bar"></span>
       <span class="icon-bar"></span>
     </div>
     <div class="list-group" style="max-width:86%">
       <a  href="/users/preferences/" class="list-group-item {% if
           'users/preferences/' in request.path%} active {% endif%}"
           ><i class="fa fa-info-circle"></i> Info</a>
       <a  href="/users/changepassword/" class="list-group-item {% if
           'users/changepassword/' in request.path%} active {%
           endif%}" ><i class="fa fa-truck"></i> Change password</a>
       <a href="/users/changeapikeys/" class="list-group-item {% if
           'users/changeapikeys/' in request.path%} active {% endif%}"
           ><i class="fa fa-key"></i> Change API keys</a></a>
     </div>
     </div>
     <div class="grid_9">
         {% block main_content %}{% endblock %}
     </div>
   </div><!-- end row-->

{% endblock %}
```

Settings template

Listing 10.13: Users preferences template.

```
{% extends 'users/users_base.html' %}
{% block title %}User preferences{% endblock %}

{% block main_content %}
  <h3>Welcome {{ user.username }} </h3> <br>
  <table>
    <tr><td style="padding: 10px;"><p><b>Email:</b></td><td>{{
        user.email }}</p></td></tr>
  <tr><td style="padding: 10px;"><p><b>API key:</b></td><td>{{
      user.userprofile.btce_key }}</p></td></tr>
  <tr><td style="padding: 10px;"><p><b>Date joined:</b></td><td>{{
      user.date_joined }}</p></td></tr>
  <tr><td style="padding: 10px;"><p><b>Last login:</b></td><td>{{
      user.last_login }}</p></td></tr>
  </table>
{% endblock %}
```

Change API key template

Listing 10.14: Change API keys template.

```
{% extends 'users/users_base.html' %}
{% block title %}Change your API keys.{% endblock %}

{% block main_content %}
  {% if error %}
     <h4><font color="red"><b>{{ error }}</b></font></h4>
  {% elif success %}
     <h4><font color="success"><b>{{ success }}</b></font></h4>
  {% endif %}
  <h3>Change your API keys:</h3> <br>
  <table>
    <form action="" method="post" >
      {% csrf_token %}
      <tr><td style="padding:10px;"><b>API key:</b></td><td>{{
          ckf.apikey }}</td><tr>
      <tr><td style="padding:10px;"><b>Secret key:</b></td><td>{{
          ckf.secretkey }}</td><tr>
      <tr><td style="padding:10px;"><input type="submit"
          name="_changepass" value="Change" class="btn
          btn-default"/></td></tr>
    </form>
  </table>
{% endblock %}
```

Change password template

Listing 10.15: Change password template.

```
{% extends 'users/users_base.html' %}
{% block title %}Change your password.{% endblock %}

{% block main_content %}
  {% if error %}
```

```
    <h4><font color="red"><b>{{ error }}</b></font></h4>
  {% elif success %}
    <h4><font color="success"><b>{{ success }}</b></font></h4>
  {% endif %}
  <h3>Change your password:</h3> <br>
  <table>
    <form action="" method="post" >
      {% csrf_token %}
      <tr><td style="padding:10px;"><b>Current
          password:</b></td><td>{{ cpf.oldpass }}</td><tr>
      <tr><td style="padding:10px;"><b>New password:</b></td><td>{{
          cpf.newpass1 }}</td><tr>
      <tr><td style="padding:10px;"><b>Repeat
          password:</b></td><td>{{ cpf.newpass2 }}</td><tr>
      <tr><td style="padding:10px;"><input type="submit"
          name="_changepass" value="Change" class="btn
          btn-default"/></td></tr>
    </form>
  </table>
{% endblock %}
```

Login template

Listing 10.16: Login template.

```
{% extends 'base.html' %}
{% block body_main %}
<section id="content" style="margin-bottom:60px; width:100%"
    class="wide-content">
    <div class="row">
        <div class="grid_3"></div>
            <div class="grid_8">
              <table>
                  <tr><td style="padding:
                      13px;"><h1>Login</h1></td></tr>
                    <form id="login_form" method="post"
                        action="/users/login/">
                        {% csrf_token %}
      <tr><td style="padding: 13px;"><b>Username:</b></td><td><input
          type="text" name="username" value="" size="30"
          style="width:200px;" class="form-control"/></td></tr>
                        <tr><td style="padding:
                            13px;"><b>Password:</b></td><td><input
                            type="password" name="password" value=""
                            size="30" style="width:200px;"
                            class="form-control"/></td></tr>

                        <tr><td style="padding: 13px;"><input
                            type="submit" value="Log in" class="btn
                            btn-default"/></td></tr>
                      </table>
                    </form>

            </div><!--end grid 8-->
        </div>
    </div>
</section>
{% endblock %}
```

URLs

Listing 10.17: Users urls.

```
from django.conf.urls import patterns, url

import views

urlpatterns = patterns('',url(r'^register/$',views.register, name
    ='register'),
                       url(r'^login/$',views.user_login, name
                           ='user_login'),
                       url(r'^logout/$',views.user_logout, name
                           ='user_logout'),
                       url(r'^preferences/$', views.preferences,
                           name='user_preferences'),
                       url(r'^changepassword/$', views.change_password,
                           name='change_password'),
                       url(r'^changeapikeys/$', views.change_api_keys,
                           name='change_api_keys'),
                       )
```

10.3.4 Wallet

Models

Listing 10.18: Wallet models.

```
from django.db import models
from users.models import UserProfile
from django.contrib.auth.models import User
import pickle
import pika
```

```python
import logging

log = logging.getLogger(__name__)

class Currency(models.Model):
    '''
    Represents a a currency.
    '''
    label = models.CharField(max_length=3)
    name = models.CharField(max_length=20, blank = True)

    def __unicode__(self):
        return unicode(self.label)


class Change(models.Model):
    '''
    Contains the name a valid btce pair which is composed by two
        currencies.
    '''
    label = models.CharField(max_length=7)

    def __unicode__(self):
        return unicode(self.label)


class Funds(models.Model):
    '''
    This class stores the total amount of coins owned by a user in a
        time.
    currency = tav -> total available.
    currency = ttl -> total (sum available and in orders).
    '''
    currency = models.CharField(max_length=3)
    amount = models.FloatField(default=0)
    datetime = models.DateTimeField(auto_now=True,blank=True)
    user = models.ForeignKey(User)

    def __unicode__(self):
        return u'%s %s %s' % (self.currency, self.amount,
            self.datetime)

class Ticker(models.Model):
    label = models.CharField(max_length=7)
    high = models.FloatField()
    low = models.FloatField()
    avg = models.FloatField()
    vol = models.FloatField()
    vol_cur = models.FloatField()
    last = models.FloatField()
    buy = models.FloatField()
    sell = models.FloatField()
    updated = models.IntegerField()
    server_time = models.IntegerField()

    def __unicode__(self):
        return u'%s %f' % (self.label, self.average)
```

Views

Listing 10.19: Wallet views.

```
from __future__ import division
from collections import Counter
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
from wallet.models import *
from django.db.models import Q
from forms import *
from btceapi import *
from orders import models as omodels
import logging
logger = logging.getLogger(__name__)


@login_required
def summary(request):
    '''
    This view is a summary of the user's info.
    Also called after the users login.
    '''
    context = {}
    sk = str(request.user.userprofile.btce_secret_key)
    ak = str(request.user.userprofile.btce_key)
    # Table with info
    info=get_info(sk, ak)
    context['info'] = info['return']

    # Grafic with funds:
    f_hist = Funds.objects.filter(user = request.user, currency =
            'ttl').order_by('-id')[:200][::-1]
    f_hist = [ [(f.datetime.year, f.datetime.month, f.datetime.day,
        f.datetime.hour, f.datetime.minute, f.datetime.second, 0),
            f.amount] for f in f_hist ]
    context['f_hist'] = f_hist

    # Grafic with number of orders
    num_orders = {}
    num_orders['Simple'] =
        len(omodels.SimpleOrder.objects.filter(user = request.user))
    num_orders['Sliced'] =
        len(omodels.SlicedOrder.objects.filter(user = request.user))
    num_orders['Paired'] =
        len(omodels.PairedOrder.objects.filter(user = request.user))
    num_orders['TimeBased'] =
        len(omodels.TimeBasedOrder.objects.filter(user =
        request.user))
    num_orders['StopLoss'] =
        len(omodels.StopLossOrder.objects.filter(user = request.user))
    context['num_orders'] = num_orders

    return render(request,'wallet/main.html', context)


@login_required
def tradeHistory(request):
    '''
    This view allows a user to search in the trade history for his
        account.
    A form is presented to filter the results. The filters are
        matching all the
    options that the btc-e API enables.
    With the information introduced by the user, the view queries the
        API
    '''
    context = {}
    context['form'] = tradeHistoryForm()
    if request.method == 'POST':
        form = tradeHistoryForm(request.POST)
        if form.is_valid():
            parameters={}
            if form.cleaned_data['nfrom']:
                parameters['from']=form.cleaned_data['nfrom']
            if form.cleaned_data['count']:
                parameters['count']=form.cleaned_data['count']
            if form.cleaned_data['from_id']:
                parameters['from_id']=form.cleaned_data['from_id']
            if form.cleaned_data['end_id']:
                parameters['end_id']=form.cleaned_data['end_id']
            if form.cleaned_data['since']:
                timestamp=int(time.mktime(form.cleaned_data['since'].timetuple()))
                parameters['since']=timestamp
            if form.cleaned_data['end']:
                timestamp=int(time.mktime(form.cleaned_data['end'].timetuple()))
                parameters['end']=timestamp
            if form.cleaned_data['pair']:
                parameters['pair']=form.cleaned_data['pair']

            resp =
                trade_history(request.user.userprofile.btce_secret_key,
                    request.user.userprofile.btce_key, parameters)
            context['resp'] = resp
    return render(request,'wallet/tradehistory.html', context)


@login_required
def transactionHistory(request):
    '''
    This view allows a user to search in his history of transaction
        in and out
    of btc-e.
    A form is presented to filter the results. The filters are
        matching all the
    options that the btc-e API enables.
    '''
    context = {}
    context['form'] = tradeHistoryForm()
    if request.method == 'POST':
        form = transactionHistoryForm(request.POST)
        if form.is_valid():
            parameters={}
            if form.cleaned_data['nfrom']:
                parameters['from']=form.cleaned_data['nfrom']
            if form.cleaned_data['count']:
                parameters['count']=form.cleaned_data['count']
            if form.cleaned_data['from_id']:
                parameters['from_id']=form.cleaned_data['from_id']
            if form.cleaned_data['end_id']:
                parameters['end_id']=form.cleaned_data['end_id']
            if form.cleaned_data['since']:
                timestamp =
                    int(time.mktime(form.cleaned_data['since'].timetuple()))
                parameters['since']=timestamp
            if form.cleaned_data['end']:
                timestamp =
                    int(time.mktime(form.cleaned_data['end'].timetuple()))
                parameters['end']=timestamp

            resp =
                transaction_history(request.user.userprofile.btce_secret_key,
                    request.user.userprofile.btce_key, parameters)
            resp = translate_transaction_status_codes(resp)
            context['resp'] = resp
    return render(request,'wallet/transhistory.html', context)
```

```


def translate_transaction_status_codes(d):
    '''
    This function maps the status codes from "type" and "status"
    returned from the btc-e API to a human readable words.
    '''
    for k,v in d.iteritems():
        if v['type'] == 5:
            d[k]['type'] = 'Debit'
        elif v['type'] == 4:
            d[k]['type'] = 'Credit'
        elif v['type'] == 2:
            d[k]['type'] = 'Withdrawal'
        elif v['type'] == 1:
            d[k]['type'] = 'Deposit'
        if v['status'] == 2:
            d[k]['status'] = 'Successful'
        elif v['status'] == 0:
            d[k]['status'] = 'Canceled/Failed'
        elif v['status'] == 1:
            d[k]['status'] = 'Waiting for aceptance'
        elif v['status'] == 0:
            d[k]['status'] = 'Not confirmed'
    return d


@login_required
def funds(request):
    '''
    Return the current user funds:
    Availables, in orders, the price in usd per currency, the amount
        in usd that represents the currency
    and the total in usd
    :param request:
    :return: dictionary with the data described above
    '''
    context = {}
    sk = request.user.userprofile.btce_secret_key
    ak = request.user.userprofile.btce_key

    funds = get_funds(sk, ak)
    funds_usd = convert_funds_to_usd(funds)
    funds_total = funds_usd.copy()
    funds_not_av = get_funds_in_active_orders(sk, ak)
    # Add missing currencies with value null to funds from active
        orders
    f_null = funds.fromkeys(funds.keys(), 0)
    funds_not_av = {x: funds_not_av.get(x, 0) + f_null.get(x, 0) for x in
            set(funds_not_av).union(f_null)}
    funds_not_av_usd = convert_funds_to_usd(funds_not_av)

    funds_total = dict(Counter(funds) + Counter(funds_not_av))
    funds_total_usd = convert_funds_to_usd(funds_total)

    context ={}
    context['f'] = funds_usd
    context['f_nav'] = funds_not_av_usd
    context['f_total'] = funds_total_usd
    return render(request, 'wallet/funds.html', context)


def convert_funds_to_usd(funds):
    '''
    Take a dict returned by btceapi.getfunds() function, convert the value of
    all the different coins into dollars and return a dict with te form:
    d[coin]['amount'] = amount
    d[coin]['usd'] =  amount_in_usd
    '''
    tickers = get_tickers()
    f = {}
    for k, v in funds.iteritems():
        f[k] = {}
        f[k]['amount'] = v
        f[k]['usd'] = convert_to_usd(k, v, tickers)
    return f


def convert_to_usd(coin, amount, tickers=None):
    '''
    Convert an amount of coin to USD using the value from the last trade done.
    It can receive the tickers values to calculate the prices from.
    If not tickets are given, is will fetch the lasts tickers
    '''
    if coin == 'usd':
        return round(amount, 4)
    else:
        if tickers == None:
            tickers = get_tickers()
        for k,v in tickers.iteritems():
            if coin in k:
                # if usd appears as a second member of xxx_xxx: Multiply
                if 'usd' in k[4:7]:
                    r = amount * v['last']
                    return round(r, 4)
                elif 'usd' in k[0:3]: # if not, divide
                    r = amount / v['last']
                    return round(r, 4)


def get_funds_in_active_orders(sk, ak):
    '''
    This function gets the amount of currency present in orders
        active.
    It returns a dict containing the values for each currency.
    '''
    #Add the funds in active orders:
    ao = get_active_orders(sk, ak)
    funds = {}
    if ao and ao['success'] == 1 and len(ao) > 0:
        for k, v in ao['return'].iteritems():
            if v['type'] == 'sell': #if sell, the currency owned is
                the first in the pair
                funds[v['pair'][0:3]] = v['amount']
            else:
                funds[v['pair'][4:7]] = v['amount']
    return funds


@login_required
def fund(request, fund):
    '''
    This view shows a graph with the historic data of a coin.
    '''
    context = {}
    f_hist = Funds.objects.filter(currency =
        fund).order_by('-id')[:200][::-1]
    f_hist = [ [(f.datetime.year, f.datetime.month, f.datetime.day,
        f.datetime.hour, f.datetime.minute, f.datetime.second, 0),
            f.amount] for
        f in f_hist ]
    context['fund'] = f_hist
    context['f'] = fund.upper()
    return render(request, 'wallet/fund.html', context)


@login_required
def reports(request):
    '''
    This view shows reports for the las 5 orders finished or every
        kind.
    The finished orders are the one with status 'executed' or
        'canceled'
    '''
    context = {}
    simple = omodels.SimpleOrder.objects.filter(user =
            request.user).filter(Q(status = 'executed') | Q(status =
                'canceled')).order_by('-id')
    if len(simple) < 6:
        simple = simple[0:len(simple)]
    else:
        simple = simple[0:5]
    sliced = omodels.SlicedOrder.objects.filter(user =
            request.user).filter(Q(status = 'executed') | Q(status =
                'canceled')).order_by('-id')
    if len(sliced) < 6:
        slc_exec = sliced[0:len(sliced)]
    else:
        slc_exec = sliced[0:5]
    sliced = {}
    for o in slc_exec:
        sliced[o.id] = {}
        sliced[o.id]['pair'] = o.pair
        sliced[o.id]['buysell'] = o.buysell
        sliced[o.id]['status'] = o.status
        sliced[o.id]['amount'] = o.amount
        sliced[o.id]['num'] = o.numberOfOrders
        sliced[o.id]['lb'] = o.lowerBound
        sliced[o.id]['ub'] = o.upperBound
        sliced[o.id]['total'] = 0
        sliced[o.id]['sp'] = {}
        for s in o.btceOrders.all():
            sliced[o.id]['sp'][s.id] = {}
            sliced[o.id]['sp'][s.id]['amount'] = s.amount
            sliced[o.id]['sp'][s.id]['price'] = s.price
            sliced[o.id]['sp'][s.id]['status'] = s.status
            sliced[o.id]['sp'][s.id]['btceid'] = s.btceid
            sliced[o.id]['total'] += s.price * s.amount

    # Using .values() for get paired orders as a dictionary
    paired = omodels.PairedOrder.objects.filter(user =
            request.user).filter(Q(status = 'executed') | Q(status =
                'canceled') | Q(status =
                    'cont_executed')).order_by('-id').values()
    if len(paired) < 6:
        paired = paired[0:len(paired)]
    else:
        paired = paired[0:5]
    # Calcule the benefit obtained with the paired order
    for p in paired:
        if p['status'] == 'cont_executed':
            # TODO get dynamicaly the fee
            total1 = p['price'] * p['amount']
            total2 = p['contraprice']* p['amount']
            comission = total1 * 0.002 + total2 * 0.002
            benefit = total2 - total1 - comission
            p['benefit'] = benefit

    time = omodels.TimeBasedOrder.objects.filter(user =
            request.user).filter(Q(status = 'executed') | Q(status =
                'canceled') | Q(status = 'expired')).order_by('-id')
    if len(time) < 6:
        time = time[0:len(time)]
    else:
        time = time[0:5]

    stop = omodels.StopLossOrder.objects.filter(user =
            request.user).filter(Q(status = 'executed') | Q(status =
                'canceled')).order_by('-id')

    context['simple'] = simple
    context['sliced'] = sliced
    context['time'] = time
    context['paired'] = paired
    return render(request,'wallet/reports.html',context)
```

Forms

Listing 10.20: Wallet forms.

```
from django import forms
from wallet.models import Change
from datetimewidget.widgets import DateTimeWidget


class tradeHistoryForm(forms.Form):
    cl=[]
    changes = Change.objects.all()
    [cl.append((c.label,c.label)) for c in changes]
    pair = forms.ChoiceField(required=False,
        widget=forms.Select(attrs =
        {'class': 'form-control', 'style':'width:130px;'}),
            choices=cl)
    nfrom = forms.IntegerField(widget=forms.TextInput(attrs =
        {'style':'width:130px;',
        'class': 'form-control',
            'style':'width:130px;'}),required=False)
```

```python
    count = forms.IntegerField(widget=forms.TextInput(attrs = {'style':'width:130px;',
        'class': 'form-control', 'style':'width:130px;'}), required=False)
    from_id = forms.IntegerField(widget=forms.TextInput(attrs = {'style':'width:130px;',
        'class': 'form-control', 'style':'width:130px;'}),required=False)
    end_id = forms.IntegerField(widget=forms.TextInput(attrs = {'style':'width:130px;',
        'class': 'form-control', 'style':'width:130px;'}),required=False)
    sort = forms.ChoiceField(required=False, widget=forms.RadioSelect,
            choices=(('ASC','newest first'),('DESC','oldest first')))
    since = forms.DateTimeField(required=False, widget=DateTimeWidget(usel10n=True,
        bootstrap_version=3, attrs={'class':'form-control',
            'style':'width:240px;'}))
    end = forms.DateTimeField(required=False, widget=DateTimeWidget(usel10n=True,
        options={'weekStart':1,'todayBtn':'true'},bootstrap_version=3, attrs={'class':'form-control'}))


class transactionHistoryForm(forms.Form):
    nfrom = forms.IntegerField(widget=forms.TextInput(attrs = {'style':'width:130px;',
        'class': 'form-control', 'style':'width:130px;'}),required=False)
    count = forms.IntegerField( widget=forms.TextInput(attrs = {'style':'width:130px;',
        'class': 'form-control', 'style':'width:130px;'}), required=False)
    from_id = forms.IntegerField( widget=forms.TextInput(attrs = {'style':'width:130px;',
        'class': 'form-control', 'style':'width:130px;'}),required=False)
    end_id = forms.IntegerField( widget=forms.TextInput(attrs = {'style':'width:130px;',
        'class': 'form-control', 'style':'width:130px;'}),required=False)
    sort = forms.ChoiceField(required=False, widget=forms.RadioSelect,
            choices=(('ASC','newest first'),('DESC','oldest first')))
    since = forms.DateTimeField(required=False, widget=DateTimeWidget(usel10n=True,
        options={'weekStart':1,'todayBtn':'true'},bootstrap_version=3))
    end = forms.DateTimeField(required=False, widget=DateTimeWidget(usel10n=True,
        options={'weekStart':1,'todayBtn':'true'},bootstrap_version=3))
```

Base template

Listing 10.21: Base template.

```
{% extends 'base.html' %}
{% block body_main %}
  <section id="content" style="margin-bottom:60px; width:100%"
      class="wide-content">
    <div class="row">
     <div class="grid_3">
   <div class="mini-submenu">
     <span class="icon-bar"></span>
       <span class="icon-bar"></span>
     <span class="icon-bar"></span>
     <span class="icon-bar"></span>
   </div>
   <div class="list-group" style="max-width:86%">
     <a  href="/wallet/summary/" class="list-group-item {% if
         'wallet/summary/' in request.path%} active {% endif%}" ><i
         class="fa fa-th-large"></i> Summary</a>
     <a  href="/wallet/funds/" class="list-group-item {% if
         'wallet/funds/' in request.path%} active {% endif%}" ><i
         class="fa fa-money"></i> Funds</a>
     <a href="/wallet/reports/" class="list-group-item {% if
         'wallet/reports/' in request.path%} active {% endif%}" ><i
         class="fa fa-line-chart"></i> Reports</a></a>
     <a href="/wallet/tradehistory/" class="list-group-item {% if
         'wallet/tradehistory/' in request.path%} active {% endif%}"
         ><i class="fa fa-book"></i> Trade's history</a>
     <a href="/wallet/transactionhistory/" class="list-group-item {%
         if 'wallet/transactionhistory/' in request.path%} active {%
         endif%}" ><i class="fa fa-list "></i> Transaction's
         history</a>
   </div>
       </div>
       <div class="grid_9">
          {% block main_content %}{% endblock %}
       </div>
    </div><!-- end row-->
    <div class="row">
      <div class="grid_3"> </div>
      <div class="grid_9">{% block table %}{% endblock %}</div>
      <div class="grid_9">{% block chart %}{% endblock %}</div>
    </div>
  </section>
{% endblock %}
```

Summary template

Listing 10.22: Wallet summary template.

```
{% extends 'wallet/wallet_base.html' %}

{% block main_content %}
{% if info %}
   <h4>Summary of your account provided by btc-e.</h4>
   <table class="table table-hover table-bordered" style="width:40%"
       summary="User summary">
       <tr><th>Other info</th><th>Value</th></tr>
       <tr><td>Active orders</td><td>{{ info.open_orders }}</td></tr>
       <tr><td>Transaction count</td><td>{{ info.transaction_count
           }}</td></tr>
       <tr><td>API rights info</td><td>{{ info.rights.info
           }}</td></tr>
       <tr><td>API rights trade</td><td>{{ info.rights.trade
           }}</td></tr>
   </table>
{% endif %}
{% if f_hist %}
    <script type="text/javascript"
        src="https://www.google.com/jsapi?autoload=
        {'modules':[{'name': 'visualization', 'version':'1',
        'packages': ['corechart']}]}"></script>
      <script type=" text/javascript">
        google.load('visualization', '1', {packages: ['corechart']});
        google.setOnLoadCallback(drawChart);
        function drawChart() {
          var data = new google.visualization.DataTable();
          data.addColumn('datetime', 'X');
          data.addColumn('number', 'Total amount of USD (including
              active orders)');
          data.addRows([
            {% for f in f_hist %}
              [new Date{{ f.0 }}, {{ f.1 }} ],
            {% endfor %}
        ]);
        var options = {
          width: 600,
          height: 450,
          hAxis: { title: 'Time', format: 'dd/MM', gridlines: {count:
              '20'}, slantedText: 'true', slantedTextAngle: '60'},
          axisTitlesPosition: 'in',
          vAxis: { title: 'USD', maxValue: '220' },
          backgroundColor: {
            stroke: "#E6E6E6",
            strokeWidth: "3"
        },
        chartArea:{left:'10%',top:'10%',width:'80%',height:'60%'},
        crosshair: { trigger: 'focus', color: 'red' },
        series: {
            1: {curveType: 'function'}
        },
        legend: {position: 'bottom'}

      };
      var chart = new google.visualization.AreaChart(
          document.getElementById('funds_chart'));
        chart.draw(data, options);
      }
    </script>
    <br>
    <h4>Total funds for the lasts 2 months.</h4>
    <div id="funds_chart"></div>
    <br>
{% endif %}
{% if num_orders %}
    <script type="text/javascript"
        src="https://www.google.com/jsapi?autoload={
        'modules':[{'name': 'visualization', 'version':'1',
        'packages':['corechart']}]}"></script>
    <script type=" text/javascript">
    google.load("visualization", "1", {packages:["corechart"]});
    google.setOnLoadCallback(drawChart);
    function drawChart() {

      var data = google.visualization.arrayToDataTable([
        ["Type of order", "Amount", { role: "style" }],
        ["Simple orders", {{ num_orders.Simple }} , "#4D944D"],
        ["Sliced orders", {{ num_orders.Sliced }} , "#FFEB99"],
        ["Paired orders", {{num_orders.Paired }} , "#FF4D4D"],
        ["Time based orders", {{ num_orders.TimeBased }} ,
            "#66A3FF"],
        ["Stop loss orders", {{ num_orders.StopLoss }} , "#66C266"]
    ]);

    var view = new google.visualization.DataView(data);
    view.setColumns([0, 1,
                    { calc: "stringify",
                      sourceColumn: 1,
                      type: "string",
                      role: "annotation" },
                    2]);

    var options = {
      width: 600,
        height: 450,
        bar: {groupWidth: "50%"},
        legend: { position: "none" },
        backgroundColor: {
          stroke: "#E6E6E6",
          strokeWidth: "3"
        },
      };
      var chart = new google.visualization.ColumnChart(
          document.getElementById('orders_chart'));
      chart.draw(data, options);
    }
      </script>
      <br>
      <h4>Total orders created.</h4>
      <div id="orders_chart"></div>
{% endif %}
{% endblock %}
```

Fund template

Listing 10.23: Fund template.

```
{% extends 'wallet/wallet_base.html' %}

{% block main_content %}
{% if fund %}
    <script type="text/javascript"
        src="https://www.google.com/jsapi?autoload=
        {'modules':[{'name': 'visualization', 'version':'1',
        'packages':['corechart']}]}"></script>
      <script type=" text/javascript">
        google.load('visualization', '1', {packages: ['corechart']});
        google.setOnLoadCallback(drawChart);
        function drawChart() {
          var data = new google.visualization.DataTable();
          data.addColumn('datetime', 'X');
          data.addColumn('number', 'Total amount of {{ f }} (including
              active orders)');
          data.addRows([
            {% for f in fund %}
              [new Date{{ f.0 }}, {{ f.1 }} ],
            {% endfor %}
        ]);
        var options = {
          width: 600,
          height: 450,
        hAxis: { title: 'Time', format: 'dd/MM', gridlines: {count:
            '20'}, slantedText: 'true', slantedTextAngle: '60'},
        axisTitlesPosition: 'in',
        vAxis: { title: '{{ f }}' },
        backgroundColor: {
          stroke: "#E6E6E6",
          strokeWidth: "3"
        },
        chartArea:{left:'10%',top:'10%',width:'80%',height:'60%'},
        crosshair: { trigger: 'focus', color: 'red' },
        series: {
            1: {curveType: 'function'}
        },
        legend: {position: 'bottom'}

      };
      var chart = new google.visualization.AreaChart(
          document.getElementById('fund_chart'));
        chart.draw(data, options);
      }
    </script>
    <br>
    <h4>Total amount of {{ f }} for the lasts 2 months:</h4><br>
    <div id="fund_chart"></div>
  {% endif %}
{% endblock %}
```

Funds template

Listing 10.24: Funds template.

```
{% extends 'wallet/wallet_base.html' %}

 {% load staticfiles %}
 {% load static %}
 {% static "" as baseUrl %}
{% block title %}Current funds in your wallet{% endblock %}

{% block main_content %}
  <table><tr><td>
  <table class="table table-bordered" data-height="400" style="width:
      auto;" summary="Funds in your wallet">
   <tr><th></th><th colspan="3">Available</th></tr>
   <tr></td><td><td>Currency</td><td> - </td><td>In USD</td></tr>
     {% for k,v in f.items %}
     <td><img src="{% get_static_prefix %}/images/{{ k}}_40x40.png"
         style="width:25px; height:25px;"></td><td><a
```

```
        href="/wallet/funds/{{ k }}/">{{ k }}</a></td><td>{{
        v.amount }}</td> <td>{{ v.usd }}</td>
      </tr>
    {% endfor %}
  </table>
  </td><td>
  <table class="table table-bordered" data-height="400" style="width:
      auto;" summary="Funds in your wallet">
    <tr><th colspan="2">In orders</th></tr>
    <tr><td> - </td><td>In USD</td></tr>
      {% for k,v in f_nav.items %}
        <td style="height:42px;">{{ v.amount }}</td> <td>{{ v.usd
            }}</td>
      </tr>
    {% endfor %}
  </table>
  </td><td>
  <table class="table table-bordered" data-height="400" style="width:
      auto;" summary="Funds in your wallet">
    <tr><th colspan="2">Total</th></tr>
    <tr><td> - </td><td>In USD</td></tr>
      {% for k,v in f_total.items %}
        <td style="height:42px;">{{ v.amount }}</td> <td>{{ v.usd
            }}</td>
      </tr>
    {% endfor %}
  </table>
  </td></tr></table>

{% endblock %}
```

Reports template

Listing 10.25: Reports template.

```
{% extends 'wallet/wallet_base.html' %}


{% block main_content %}
  {% if simple %}
    <h4>Showing the last 5 simple orders</h4>
    <table class="table table-bordered table-striped"
        data-height="400" style="width: auto;" summary="Reports for
        simple orders">
  <thead>
  <tr><th class="col-md-1">Id</th><th class="col-md-2">Pair</th><th
      class="col-md-1">Type</th><th class="col-md-2">Amount</th><th
    class="col-md-2">Price</th><th class="col-md-2">Status</th></tr>
  </thead>
  <tbody>
    {% for o in simple %}
            <tr><td>{{ o.btceid }}</td><td>{{ o.pair }}</td><td>{{
                o.buysell }}</td><td>{{ o.amount }}</td><td>{{
                o.price }}</td><td>{{ o.status }}</td></tr>
          {% endfor %}
  </tbody>
     </table>
{% else %}
  <p><h4>You do not have simple orders completed.</h4></p>
{% endif %}
<br>
{% if sliced %}
<h4>Showing the last 5 sliced orders</h4>
{% for k,v in sliced.items %}
  <h5>Virtual order:</h5>
    <table class="table table-bordered table-striped"
        data-height="400" style="width: auto;" summary="Reports for
        virtual order">
    <thead>
        <tr><th class="col-md-1">Id</th><th
            class="col-md-1">Pair</th><th
            class="col-md-1">Type</th><th
            class="col-md-1">Amount</th><th class="col-md-1">#
            orders</th><th class="col-md-1">Lower bound</th><th
            class="col-md-1">Upper bound</th><th
            class="col-md-1">Total</th><th
            class="col-md-2">Status</th></tr>
    </thead>
    <tbody>
    <tr><td>{{ k }}</td><td>{{ v.pair }}</td><td>{{ v.buysell
        }}</td><td>{{ v.amount }}</td><td>{{ v.num }}</td><td>{{
        v.lb }}</td><td>{{ v.ub }}</td><td>{{ v.total }}</td><td>{{
        v.status }}</td></tr>
      </tbody>
      </table>
   <table class="table table-bordered table-striped"
       data-height="400" style="width: auto;" summary="Reports for
       real orders">
        <tr><th>Id</th><th>Amount</th><th>Price</th><th>Status</th></tr>
        <h5>Real orders:</h5>
      {% for k,o in v.sp.items %}
      <tr><td>{{ o.btceid }}</td><td>{{ o.amount }}</td><td>{{
          o.price }}</td><td>{{ o.status }}</td></tr>
        {% endfor %}
   </tbody>
     </table><br>
   {% endfor %}
{% else %}
  <p><h4>You do not have sliced orders completed.</h4></p>
{% endif %}
<br>


{% if time %}
<h4>Showing the last 5 time based orders</h4>
  <table class="table table-bordered table-striped"
      data-height="400" style="width: auto;" summary="Reports for
      time based orders">
  <thead>
      <tr><th class="col-md-1">Id</th><th
          class="col-md-1">Pair</th><th
          class="col-md-1">Type</th><th
          class="col-md-1">Amount</th><th
          class="col-md-1">Price</th><th class="col-md-3">Expiration
          time</th><th class="col-md-1">Status</th></tr>
  </thead>
  <tbody>
    {% for o in time %}
    <tr><td>{{ o.btceid }}</td><td>{{ o.pair }}</td><td>{{
        o.buysell }}</td><td>{{ o.amount }}</td><td>{{ o.price
        }}</td><td>{{ o.expiration_time }}</td><td>{{ o.status
        }}</td></tr>
      {% endfor %}
   </tbody>
     </table>
{% else %}
  <p><h4>You do not have time based orders completed.</h4></p>
{% endif %}
<br>

{% if paired %}
<h4>Showing the last 5 paired orders</h4>
  <table class="table table-bordered table-striped"
      data-height="400" style="width: auto;" summary="Reports for
      paired orders">
  <thead>
      <tr><th class="col-md-1">Id</th><th
          class="col-md-1">Pair</th><th
          class="col-md-1">Type</th><th
          class="col-md-1">Amount</th><th
          class="col-md-1">Price</th><th class="col-md-2">Contra
          price</th><th class="col-md-1">Benefit</th><th
          class="col-md-1">Status</th></tr>
  </thead>
  <tbody>
```

```
     {% for o in paired %}
     <tr><td>{{ o.id }}</td><td>{{ o.pair }}</td><td>{{ o.buysell
         }}</td><td>{{ o.amount }}</td><td>{{ o.price }}</td><td>{{
         o.contraprice }}</td><td>{{ o.benefit }}</td><td>{{
         o.status }}</td></tr>
       {% endfor %}
    </tbody>
      </table>
  {% else %}
    <p><h4>You do not have paired orders completed.</h4></p>
  {% endif %}

  <br>

  {% if stoploss %}
  {% else %}
    <p><h4>You do not have stop loss orders completed.</h4></p>
  {% endif %}
  <br><h5>Note: A value of 0 in btc order id means that the order was
      completely filled when it was created.</h5>
{% endblock %}
```

Trade history template

Listing 10.26: Trade history template.

```
{% extends 'wallet/wallet_base.html' %}

{% block header %}
{% if form %}{{ form.media }}{% endif %}
{% endblock %}

{% load timetags %}

{% block main_content %}
  {% if form %}
    <form action="" method="post">
     {% csrf_token %}
     <table>
       <tr><td style="padding:5px 17px;"><p><b>Select the
           pair:</b></p><p> {{ form.pair }}</p></td><td><p><b># trades
           to display:</p><p> {{ form.count }}</b></p></td></tr>
       <tr><td style="padding:0px 17px;"><p><b>Since id:</b></p><p>
           {{ form.from_id }}</p></td>
       <td><p><b>Until id:</b> </p><p> {{ form.end_id }}</p></td></tr>
     </table>
     <table>
```

```
      <tr><td style="padding:0px 17px;"><p><b>Since day:</b></p><p>
          {{ form.since }}</p></td><td><p><b>Until day:</b></p><p> {{
          form.end }}</p></td></tr>
      <tr><td style="padding:10px 17px;"><p><input type="submit"
          value="Get history" name="_gethistory" class="btn
          btn-default"/></p></td></tr>
    </table>
   </form>
  {% endif %}
{% endblock %}

{% block table %}
  {% if resp %}
   <table class="table table-hover table-bordered" style="width:
       auto;" summary="Trade history">
     <tr><th>Id</th><th>Pair</th><th>Amount</th>
         <th>Type</th><th>Rate</th><th>Your
         order</th><th>Time</th></tr>
     {% for k,t in resp.items %}
     {% ifequal t.type 'buy' %}<tr style="background-color: rgba(0,
         255, 0, 0.22);">{% endifequal %}
     {% ifequal t.type 'sell' %}<tr style="background-color:
         rgba(255, 0, 0, 0.22);">{% endifequal %}

     <td>{{ t.order_id }}</td><b><td>{{ t.pair }}</td></b><td>{{
         t.amount }}</td><td>{{ t.type }}</td><td>{{ t.rate
         }}</td><td>{{ t.is_your_order }}</td><td>{{
         t.timestamp|datefromstamp|date:'d/m/y h:m' }}</td></tr>
     {% endfor %}
   </table>
  {% endif %}
{% endblock %}
```

Transaction history template

Listing 10.27: Transaction history template.

```
{% extends 'wallet/wallet_base.html' %}

{% block header %}
{% if form %}{{ form.media }}{% endif %}
{% endblock %}
{% load timetags %}

{% block main_content %}
  {% if form %}
   <form action="" method="post">
```

```
    {% csrf_token %}
    <table>
      <tr><td style="padding:5px 17px;"><p><b># trades to
          display:</b><p><p> {{ form.count }}</p></td></tr>
      <tr><td style="padding:0px 17px;"><p><b>Since id:</b><p><p> {{
          form.from_id }}</p></td>
      <td><p><b>Until id:</b><p><p> {{ form.end_id }} </p></td></tr>
    </table>
    <table>
      <tr><td style="padding:0px 17px;"><p><b>Since day:</b><p><p>
          {{ form.since }}</p></td><td><p><b>Until day:</b><p><p> {{
          form.end }}</p></td></tr>
      <tr><td style="padding:10px 17px;"><p><input type="submit"
          value="Get history" name="_gethistory" class="btn
          btn-default"/></p></td></tr>
    </table>
   </form>
  {% endif %}
{% endblock %}

{% block table %}
  {% if resp %}
   <table class="table table-hover table-bordered" style="width:
       auto;" summary="Transaction history">
     <tr><th>Type</th><th>Amount</th>
         <th>Currency</th><th>Desc</th><th>Status</th><th>Time</th></tr>
     {% for k,t in resp.items %}
     <tr><td>{{ t.type }}</td><td>{{ t.amount }}</td><td>{{
         t.currency }}</td><td>{{ t.desc }}</td><td>{{ t.status
         }}</td><td>{{ t.timestamp|datefromstamp|date:'d/m/y h:m'
         }}</td></tr>
     {% endfor %}
   </table>
  {% endif %}
{% endblock %}
```

URLs

Listing 10.28: Wallet URLs.

```
from django.conf.urls import patterns, url, include

import views

urlpatterns = patterns('',
                     url(r'^summary/$', views.summary,
                         name='summary'),
                    url(r'^funds/$', views.funds, name='funds'),
                    url(r'^funds/(?P<fund>[a-z]{3})/$',
                        views.fund,name='fund'),
                    url(r'^tradehistory/$', views.tradeHistory,
                        name='tradeHistory'),
                    url(r'^transactionhistory/$',
                        views.transactionHistory,
                        name='transactionHistory'),
                    url(r'^reports/$', views.reports,
                        name='reports'),
                    url(r'^users/$', include('users.urls')),
                    )
```

10.3.5 Orders

Models

Listing 10.29: Order models.

```
from __future__ import division
from django.db import models
from django.contrib.auth.models import User
from users.models import UserProfile
from wallet.models import Change
import pickle
import pika
import logging

log = logging.getLogger('orders')

class BaseOrder(models.Model):
    '''
    BaseOrder is the base class for all the other order classes.

    Attributes:
        timestamp: UNIX timestamp generated in the moment of saving
            it into the
        db.
        user:     foreign key to the User model.
        status:   string representing the position of the order.
    '''
    timestamp = models.DateTimeField(auto_now=True,blank=True)
    user = models.ForeignKey(User)
    status = models.CharField(max_length = 20)

    @classmethod
    def create(cls, user, status):
        ''' Creates an order object saving it into the db'''
        order = cls(user=user, status=status)
        return order

    @property
    def execute(self):
        ''' Executes an order'''
        raise NotImplementedError("Subclasses should implement this!")

    @property
    def cancel(self):
        ''' Cancels an order '''
        raise NotImplementedError("Subclasses should implement this!")

    @property
    def __str__():
        ''' Reader friendly representation '''
        raise NotImplementedError("Subclasses should implement this!")


class BtceOrder(BaseOrder):
    '''
    Class containing attributes and methods to represent a valid
        btc-e order

    Attributes:
        buysell: String with possible values 'buy' || 'sell'
        pair:   Valid btc-e pair
        amount: Quantity of coins to exchange
        status: Can be 'created' || 'started' || 'canceled' ||
            'executed'
        btceid: Id returned by btce.
    '''
    pair = models.ForeignKey(Change)
    amount = models.FloatField()
    price = models.FloatField()
    buysell = models.CharField(max_length = 4) # allowed values:
        'buy' or 'sell'
    btceid = models.IntegerField(null = True)

    @classmethod
    def create(cls, pair, user, buysell, amount, price):
        order = cls(pair=pair, amount=amount, price=price,
            buysell=buysell,
                user=user, status='created')
        order.save()
        return order

    def execute(self):
        from wallet.btceapi import create_order
        log.debug('Execute: %s %s %s %s',self.buysell,self.amount,self.pair,self.price)
        r = create_order(self.user.userprofile.btce_secret_key,
                self.user.userprofile.btce_key, self.pair.label, self.buysell, self.price,self.amount)
        if 'error' in r:
            log.error('The order could not be created. Message: %s',r['error'])
            return r, -1
        else:
            self.btceid = r['return']['order_id']
            # If btceid is 0 means that the order was filled
            if self.btceid == 0:
                self.status = 'executed'
                self.save()
                log.info('Created and executed Simple order')
                return r, 0
            self.status = 'started'
            self.save()
            log.info('Created order %s', self.btceid)
            return r, 0

    def cancel(self):
        from wallet.btceapi import cancel_order
        log.info('Canceling an simple order')
        r = cancel_order(self.user.userprofile.btce_secret_key, self.user.userprofile.btce_key,
        self.btceid)
        if 'error' in r:
            log.error('The order %s could not be canceled',self.btceid)
            return r, -1
        else:
            self.status = 'canceled'
            self.save()
            log.info('The order %s has been canceled',self.btceid)
            return r, 0

    def __str__(self):
        return unicode('%s %s %s %s' % (self.buysell, self.pair.label,
            self.amount, self.price))


class SimpleOrder(BtceOrder):
    '''
    This class is exactly the same as BtceOrder.

    The purpose of this class is to separate in the database level, BtceOrder classes used by many
    other classes to SimpleOrder classes created from the frontend.
    ’’’
    pass


class SlicedOrder(BaseOrder):
    ’’’
    Class containing attributes and methods to store a sliced order in the
    db.
    A sliced order canceled will have all the suborders canceled or executed.
    If the creation of the class fails, it will be canceled.

    Attributes:
        simpleOrder: Foreigng key to a simple order. It will contain as many
        foreign keys as number of orders.
        upperBound: Maximum amount of money to pay in a order.
        lowerBound: Minimum amount of money to pay in a cancelOrder.
        numberOfOrders: Number of total orders contained in an sliced order.
        status: Can be ’created’ || ’started’ || ’canceled’ || ’executed’
    ’’’
    pair = models.ForeignKey(Change)
    amount = models.FloatField()
    buysell = models.CharField(max_length = 4) # allowed values: ’buy’ or ’sell’
    numberOfOrders = models.IntegerField()
    upperBound = models.FloatField()
    lowerBound = models.FloatField()
    btceOrders = models.ManyToManyField(’BtceOrder’, related_name=’sliced_into_order’,null=True,blank=True)

    @classmethod
    def create(cls, user, no, ub, lb, pair, amount, bs):
        if lb >= ub:
            log.error(’The upper bound must be grater than the lower bound’)
            return -1
        sliced = cls(status=’created’, user=user, numberOfOrders=no, upperBound=ub,
                lowerBound=lb, pair=pair, amount=amount, buysell=bs)
        sliced.save()
        return sliced

    def execute(self):
        ''' Creates and executed a several simple orders to compose a SlicedOrder'''
        log.info('Executing sliced order')
        step = (self.upperBound - self.lowerBound) / self.numberOfOrders
        amountPerOrder = self.amount / self.numberOfOrders
        import numpy as np
        for price in (np.arange(self.lowerBound,self.upperBound,step)):
           # TODO for truncate take the maximum number of digits allowed for each pair
           price = round(price,3)
           sp=BtceOrder.create(self.pair, self.user, self.buysell,
                  amountPerOrder, price)
           r, rc = sp.execute()
           if rc == -1:
               log.error('A problem occurred while starting a sliced order, all\
               the suborders have been canceled: %s', r)
               self.cancel()
               return r, rc
           sp.save()
           self.btceOrders.add(sp)
        self.status = 'started'
        self.save()
        return r, 0

    def cancel(self):
        ''' Cancels all the btceOrders and set its status to canceled '''
        for order in self.btceOrders.all():
            if order.status == 'started':
                r,s = order.cancel()
                if s == -1:
                    log.warning('Problem while canceling btce order %s',order.btceid)
        self.status = 'canceled'
        self.save()

    def numActive(self):
        '''
        Returns the number of suborder actives
        '''
        num = 0
        for order in self.btceOrders.all():
            if order.status == 'started':
                num += 1
        return num

    def numExecuted(self):
        '''
        Returns the number of suborder executed
        '''
        num = 0
        for order in self.btceOrders.all():
            if order.status == 'executed':
                num += 1
        return num

    def totalAmount(self):
        '''
        Returns the total amount of currency trade
        '''
        return self.upperBound * self.lowerBound

    def __str__(self):
        ''' Reader friendly representation '''
        return u'#%s %s %s %s %s %s ' %(self.numberOfOrders,
            self.buysell,
                self.pair.label, self.amount, self.upperBound,
                self.lowerBound)


class TimeBasedOrder(BtceOrder):
    '''
    Class containing attributes and methods to create a time based
        order.

    The execute method creates a TimerBaseOrder object and stores it
        in the database.
    After it sends a message with the object narrowed to the backend.
        The
    backend will cancel the order if the expiration time arrives and
        the order
    is still active.

    The cancel method will cancel a timer order making a call to the
        btce API

    status: 'created' || 'started' || 'canceled' || 'executed' ||
        'expired'
    '''
    expiration_time = models.DateTimeField()

    @classmethod
    def create(cls, pair, user, buysell, amount, price, exptime):
        order = cls(pair=pair, amount=amount, price=price,
            buysell=buysell,
                user=user, status='created', expiration_time=exptime)
        order.save()
        return order

    def execute(self):
        '''
        This method creates an order in the market and sets up a time
        limit when the order will be canceled if was not executed or
        canceled before
        '''
        from wallet.btceapi import create_order
        r = create_order(self.user.userprofile.btce_secret_key,
                self.user.userprofile.btce_key,
                    self.pair,self.buysell,self.price,self.amount)
        if 'error' in r:
            log.error('A problem occurs when starting time based
                order: %s',r)
            return r, -1
        else:
            self.btceid = r['return']['order_id']
            # If btceid is 0 means that the order was filled
            if self.btceid == 0:
                self.status = 'executed'
                self.save()
                log.info('Created and executed order')
                return r, 0
            self.status = 'started'
            self.save()
            log.info('Time based order created successfully:
                %s',self.btceid)
            send_time_based_order(self)
            return r, 0

    def cancel(self):
        '''
        This method cancels a time based order.
        It set the status to 'canceled' and the backend will delete it
        '''
        from wallet.btceapi import cancel_order
        r = cancel_order(self.user.userprofile.btce_secret_key,
            self.user.userprofile.btce_key,
        self.btceid)
        if 'error' in r:
            log.error('A problem occurs when canceling a time based
                order: %s',r)
            return r
        else:
            self.status = 'canceled'
            self.save()
```

```python
            log.info('Order canceled successfully: %s',self.btceid)
            return r

    def __str__(self):
        return u'%s %s %s %s' % (self.buysell, self.pair.label, self.amount,
                self.price, self.expiration_time)


def send_time_based_order(order):
    '''
    This function sends a TimeBasedOrder to its correspondent queue where it
    will be handled by the the backend
    '''
    sl = pickle.dumps(order)
    con = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
    chan = con.channel()
    chan.basic_publish(exchange = '', routing_key='order_timer', body=sl,properties=pika.BasicProperties(delivery_mode = 2,))


class PairedOrder(BtceOrder):
    '''
    This class represents a paired order.
    A paired order will be executed when both orders has been executed.
    Its status will be canceled if one of the two orders has been canceled.

    status: 'created' || 'started' || 'canceled' || 'executed' ||
    'cont_started' || 'cont_canceled' || 'cont_executed'
    '''
    contraprice = models.FloatField(null=True)
    contrabuysell = models.CharField(max_length = 4,null=True)
    contrabtceid = models.IntegerField(null = True)
    @classmethod
    def create(cls, pair, user, buysell, amount, price, cp=None):
        cbs = 'sell' if buysell == 'buy' else 'buy'
        order = cls(pair=pair, amount=amount, price=price, buysell=buysell,
                user=user, status='created', contrabuysell= cbs)
        order.save()
        return order

    def execute_contraorder(self):
        from wallet.btceapi import create_order
        log.debug('Executing contraorder: %s %s %s
            %s',self.buysell,self.amount,self.pair,self.price)
        r = create_order(self.user.userprofile.btce_secret_key,
                self.user.userprofile.btce_key, self.pair.label,
                self.contrabuysell, self.contraprice, self.amount)
        if 'error' in r:
            log.error('The order could not be created. Message:
                %s',r['error'])
            return r, -1
        else:
            self.contrabtceid = r['return']['order_id']
            # If btceid is 0 means that the order was filled
            if self.contrabtceid == 0:
                self.status = 'cont_executed'
                self.save()
                log.info('Created and executed order')
                return r, 0
            self.status = 'cont_started'
            self.save()
            log.info('Created contraorder %s', self.btceid)
            return r, 0

    def cancel(self):
        from wallet.btceapi import cancel_order
        log.info('Canceling a paired order')
        if self.btceid is not None:
            r = cancel_order(self.user.userprofile.btce_secret_key,
                self.user.userprofile.btce_key,
            self.btceid)
            if 'error' in r:
                log.error('The order %s could not be
                    canceled',self.btceid)
                return r, -1
        if self.contrabtceid is not None:
            r = cancel_order(self.user.userprofile.btce_secret_key,
                self.user.userprofile.btce_key,
            self.contrabtceid)
            if 'error' in r:
                log.error('The order %s could not be
                    canceled',self.btceid)
                return r, -1
        self.status = 'canceled'
        self.save()
        log.info('The paired order %s has been canceled',self.btceid)
        return r, 0

    def __unicode__(self):
        return u'%s %s %s' % (self.pair, self.amount, self.buysell)
```

```

class StopLossOrder(BtceOrder):
    '''
    Class containing attributes and methods to create a stop loss order, manage
    it and store it in the db.

    Attributes:
        stoploss: Amount at which sell automatically
        status: 'created' || 'started' || 'canceled' || 'executed'
    '''

    def _send_to_queue(self):
        '''
        Sends itself to the stop loss queue to be picked
        and managed by the stop loss backend
        '''
        return 0

    def _cancel_from_queue(self):
        '''
        Send a message to the queue to cancel itself.
        '''
        return 0

    def execute(self):
        from wallet.btceapi import create_order
        log.debug('Adding stop loss order to the market: %s %s %s %s',self.buysell,self.amount,self.pair,self.price)
        r = self._send_to_queue()
        if r == 0:
            self.status = 'executed'
            self.save()
            log.info('Created stop loss order %s', self.btceid)
            return 'success', 0
        else:
            self.status = 'canceled'
            self.save()
            log.error('The stop loss order could not be created. Message: %s',r['error'])
            return r, -1

        return r, 0

    def cancel(self):
        from wallet.btceapi import cancel_order
        self._cancel_from_queue()
        if r ==0:
            self.status = 'started'
            self.save()
            log.info('The stop loss order %s has been
                canceled',self.btceid)
            return 'success', 0
        else:
            log.error('The stop loss order %s could not be
                canceled',self.btceid)
            return r, -1

    def __str__(self):
        return u'%s %s %s %s %s' % (self.buysell, self.pair.label,
            self.amount,
                self.price, self.stoploss)
```

Views

Listing 10.30: Order views.

```
from __future__ import division
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
from django.db.models import Q
from orders import forms
from orders.models import *
from wallet.btceapi import *
import time
import pika, pickle
import logging
logger = logging.getLogger(__name__)

@login_required
def simpleOrder(request):
    '''
    When the method is GET it renders the web page displaying: the
        left
    navigation menu, a table with the available funds, and the
        simpleOrderFormto
    create a simple order.
    When the method is POST it validates the form, creates a simple
        order and
    renders again the page of before showing a message with the
        result of the
    order creation.
    '''
    context = {}
    if request.method == 'POST':
        form = forms.simpleOrderForm(request.POST)
```

```
       if form.is_valid():
           sp=SimpleOrder.create(form.cleaned_data['pair'],
               request.user, form.cleaned_data['buysell'],
                   form.cleaned_data['amount'],
                       form.cleaned_data['price'])
           resp=sp.execute()
           print resp
           if resp[1] == -1:
               context['error'] = resp[0]['error']
           else:
               context['success'] = 'Order created successfully'
       else:
           context['error'] = form.errors
    context['funds'] =
        get_funds(request.user.userprofile.btce_secret_key,
           request.user.userprofile.btce_key, nround=4)
    context['form'] = forms.simpleOrderForm()
    return render(request,'orders/simpleorder.html',context)



@login_required
def slicedOrder(request):
     """
     Renders the sliced order page and accepts POST request for
         creating sliced
     orders.

     """
     context = {}
     if request.method == 'POST':
        form = forms.slicedOrderForm(request.POST)
        if form.is_valid():
            f = form.cleaned_data
            slo = SlicedOrder.create(request.user,
                f['numberOfOrders'], f['upperBound'],
                    f['lowerBound'],f['pair'],f['amount'],f['buysell'])
            resp = slo.execute()
            if resp[1] == -1:
                context['error'] = resp[0]['error']
            else:
                context['success'] = 'Created %s orders successfully'
                    % \
                f['numberOfOrders']
        else:
            context['error'] = form.errors
     context['form'] = forms.slicedOrderForm()
     context['funds'] =
         get_funds(request.user.userprofile.btce_secret_key,
            request.user.userprofile.btce_key, nround=4)
    return render(request,'orders/slicedorder.html',context)


@login_required
def timeBasedOrder(request):
    """
    Renders the timer order page and accepts POST request creating
        time based
    orders.

    """
    context = {}
    if request.method == 'POST':
       form = forms.timeBasedOrderForm(request.POST)
       if form.is_valid():
           f = form.cleaned_data
           to = TimeBasedOrder.create(f['pair'], request.user,
                   f['buysell'], f['amount'], f['price'],
                       f['expiration_time'])
           resp = to.execute()
           if resp[1] == -1:
               context['error'] = resp[0]['error']
           else:
               context['success'] = 'Order created successfully'
       else:
           context['error'] = form.errors
    context['funds'] =
        get_funds(request.user.userprofile.btce_secret_key
            ,request.user.userprofile.btce_key, nround=4)
    context['form'] = forms.timeBasedOrderForm()
    return render(request,'orders/timebasedorder.html',context)


@login_required
def pairedOrder(request):
    """
    Renders the paired order page and accepts POST request creating
        paired
    orders.

    """
    context = {}
    if request.method == 'POST':
       if request.POST.get("_order"):
           form = forms.pairedOrderForm(request.POST)
           if form.is_valid():
               f = form.cleaned_data
               po = PairedOrder.create(f['pair'], request.user,
                        f['buysell'], f['amount'], f['price'])
                resp = po.execute()
                if resp[1] == -1:
                    context['error'] = resp[0]['error']
                else:
                    context['success'] = 'Order created successfully'
            else:
                context['error'] = form.errors
        elif request.POST.get("_contraOrder"):
            form = forms.contraOrderForm(request.POST, user = request.user)
            if form.is_valid():
                f = form.cleaned_data
                po = PairedOrder.objects.filter(user =
                        request.user).filter(id = f['opened'].id)[0]
                po.contraprice = f['contraprice']

                resp = po.execute_contraorder()
                if resp[1] == -1:
                    context['error'] = resp[0]['error']
                else:
                    context['success'] = 'Contra order created successfully'
            else:
                context['error'] = form.errors
    context['funds'] = get_funds(request.user.userprofile.btce_secret_key
            ,request.user.userprofile.btce_key, nround=4)
    context['orderForm'] = forms.pairedOrderForm()
    context['contraOrderForm'] = forms.contraOrderForm(user = request.user)
    return render(request,'orders/pairedorder.html',context)


@login_required
def stopLossOrder(request):
    """
    Renders the stop loss order page and accepts POST request creating stop
    loss orders.

    """
    context = {}
    if request.method == 'POST':
        form = forms.stopLossOrderForm(request.POST)
        if form.is_valid():
            f = form.cleaned_data
            to = StopLossOrder.create(f['pair'], request.user,
                    f['amount'], f['price'], f['stoploss'])
            resp = to.execute()
            if resp[1] == -1:
                context['error'] = resp[0]['error']
            else:
                context['success'] = 'Order created successfully'
        else:
            context['error'] = form.errors
    context['funds'] = get_funds(request.user.userprofile.btce_secret_key
        ,request.user.userprofile.btce_key, nround=4)
    context['form'] = forms.stopLossOrderForm()
    return render(request,'orders/stoplossorder.html',context)


@login_required
def activeOrders(request):
    '''
    Render a page with a user's active orders.

    Each order can be canceled from this page.
    Each order shows its different attributes agrupted by column.
    '''
    simple = SimpleOrder.objects.filter(user=request.user).filter(status =
        'started')
    sliced = SlicedOrder.objects.filter(user = request.user).filter(status =
        'started')
    timed = TimeBasedOrder.objects.filter(user = request.user).filter(status =
        'started')
    paired = PairedOrder.objects.filter(user = request.user).filter(~Q(status =
        'canceled') & ~Q(status = 'cont_executed'))
    stoploss = StopLossOrder.objects.filter(user = request.user).filter(status =
        'started')

    simp = {}
    slic = {}
    pair = {}
    time = {}
    stop = []
    for o in simple:
        id = str(o.btceid)
        simp[id] = {}
        simp[id]['pair'] = o.pair.label
        simp[id]['amount'] = o.amount
        simp[id]['type'] = o.buysell
        simp[id]['price'] = o.price
        simp[id]['total'] = o.amount * o.price
    for o in sliced:
        id = str(o.id)
        slic[id] = {}
        slic[id]['pair'] = o.pair.label
        slic[id]['amount'] = o.amount
        slic[id]['type'] = o.buysell
        slic[id]['number'] = o.numberOfOrders
        slic[id]['upper'] = o.upperBound
        slic[id]['lower'] = o.lowerBound
        slic[id]['total'] = o.totalAmount()
        slic[id]['active'] = o.numActive()
        slic[id]['executed'] = o.numExecuted()
    for o in paired:
        id = str(o.id)
        pair[id] = {}
        pair[id]['pair'] = o.pair.label
        pair[id]['amount'] = o.amount
        pair[id]['type'] = o.buysell
        pair[id]['price'] = o.price
        pair[id]['total'] = o.amount * o.price
        pair[id]['cprice'] = o.contraprice
    for o in timed:
        id = str(o.id)
        time[id] = {}
        time[id]['pair'] = o.pair.label
        time[id]['amount'] = o.amount
        time[id]['type'] = o.buysell
        time[id]['price'] = o.price
        time[id]['total'] = o.amount * o.price
        time[id]['exp'] = o.expiration_time
    for o in stoploss:
        id = str(o.btceid)
        stop[id] = {}
        stop[id]['pair'] = o.pair.label
        stop[id]['amount'] = o.amount
        stop[id]['type'] = o.buysell
        stop[id]['price'] = o.price
        stop[id]['total'] = o.amount * o.price

    context={'simple': simp, 'sliced' : slic , 'paired': pair,
        'time': time}
    return render(request,'orders/activeorders.html',context)


@login_required
def cancelSimpleOrder(request,order_id):
    '''
    It cancels a simple active order
    '''
    order =
        SimpleOrder.objects.filter(btceid=order_id,user=request.user)[0]
    order.cancel()
    return activeOrders(request)


@login_required
def cancelSlicedOrder(request,order_id):
    '''
    It cancels a simple active order
    '''
    order =
        SlicedOrder.objects.filter(id=order_id,user=request.user)[0]
    order.cancel()
    return activeOrders(request)


@login_required
def cancelPairedOrder(request,order_id):
    '''
    It cancels a paired order
    '''
    order =
        PairedOrder.objects.filter(id=order_id,user=request.user)[0]
    order.cancel()
    return activeOrders(request)

@login_required
def cancelTimeBasedOrder(request,order_id):
    '''
    It cancels a time based order
    '''
    order =
        TimeBasedOrder.objects.filter(id=order_id,user=request.user)[0]
    order.cancel()
    return activeOrders(request)

@login_required
def cancelStopLossOrder(request,order_id):
    '''
    It cancels a stop loss order
    '''
    order =
        StopLossOrder.objects.filter(id=order_id,user=request.user)[0]
    order.cancel()
    return activeOrders(request)
```

Forms

Listing 10.31: Order forms.

```
from django import forms
from wallet.models import Change as wChange
from orders.models import *
from datetimewidget.widgets import DateTimeWidget


class simpleOrderForm(forms.Form):
    pair = forms.ModelChoiceField(queryset=wChange.objects.all(),
            empty_label="Select a pair",
                widget=forms.Select(attrs={'class': 'form-control',
                'style':'width:130px;'}),required=True)
    amount = forms.FloatField(widget=forms.TextInput(
        attrs={'style':'width:130px;',
        'class':'form-control'}),required=True)
    price = forms.FloatField(widget=forms.TextInput(
        attrs={'style':'width:130px;',
        'class':'form-control'}),required=True)
    buysell = forms.ChoiceField(required=True,
        widget=forms.RadioSelect,
        choices=(('buy','buy',),('sell','sell',)))


class slicedOrderForm(forms.Form):
    pair = forms.ModelChoiceField(queryset=wChange.objects.all(),
            empty_label="Select a pair",
                widget=forms.Select(attrs={'class': 'form-control',
                'style':'width:130px;'}))
    amount = forms.FloatField(widget=forms.TextInput(
        attrs={'style':'width:130px;',
        'class':'form-control'}),required=True)
    numberOfOrders = forms.IntegerField( widget=forms.TextInput(
        attrs={'style':'width:130px;',
        'class':'form-control'}),required=True)
    lowerBound = forms.FloatField(widget=forms.TextInput(
        attrs={'style':'width:130px;',
        'class':'form-control'}),required=True)
    upperBound = forms.FloatField(widget=forms.TextInput(
        attrs={'style':'width:130px;',
        'class':'form-control'}),required=True)
    buysell= forms.ChoiceField(required=True,
        widget=forms.RadioSelect,
        choices=(('buy','buy',),('sell','sell',)))


class timeBasedOrderForm(simpleOrderForm):
    expiration_time = forms.DateTimeField(required=True, widget =
            DateTimeWidget(usel10n = True,bootstrap_version=3,
                attrs={'class':'form-control'}))


class pairedOrderForm(simpleOrderForm):
    '''
    This form is used to create the first of the two paired orders
    '''
    pass


class contraOrderForm(forms.Form):
    '''
    This form allows a user to create the contra order of a paired
        order. Note
    that the first order have to be already executed
    '''
    def __init__(self,*args,**kwargs):
        user = kwargs.pop('user')
        super(contraOrderForm,self).__init__(*args,**kwargs)
        self.fields['opened'] = forms.ModelChoiceField(queryset =
            PairedOrder.objects.filter(user =
            user).filter(status = 'executed'), empty_label="Parent
                order", widget=forms.Select(attrs={'class':
                'form-control', 'style':'width:130px;'}),required=True)
        self.fields['contraprice'] = \
        forms.FloatField(widget=forms.TextInput(attrs =
            {'style':'width:130px;',
            'class':'form-control'}),required=True)

    opened = forms.MultipleChoiceField()
    contraprice = forms.FloatField()


class stopLossOrderForm(simpleOrderForm):
    pass
```

Base order template

Listing 10.32: Base order template.

```
{% extends 'base.html' %}
{% block body_main %}
 {% load staticfiles %}
 {% load static %}
 {% static "" as baseUrl %}
```

```
<section id="content" style="margin-bottom:60px; width:100%"
    class="wide-content">
  <div class="row">
    <div class="grid_3"><!-- aside menu content -->
      <div class="mini-submenu">
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </div>
      <div class="list-group" style="max-width:86%">
        <a  href="/orders/activeorders/" class="list-group-item {%
            if 'orders/activeorders/' in request.path%} active {%
            endif%}" ><i class="fa fa-play"></i> Active orders</a>
        <a href="/orders/simpleorder/" class="list-group-item {% if
            'orders/simpleorder/' in request.path%} active {%
            endif%}" ><i class="fa fa-circle-o"></i> Simple
            order</a></a>
        <a href="/orders/slicedorder/" class="list-group-item {% if
            'orders/slicedorder/' in request.path%} active {%
            endif%}" ><i class="fa fa-signal"></i> Sliced order</a>
        <a href="/orders/pairedorder/" class="list-group-item {% if
            'orders/pairedorder/' in request.path%} active {%
            endif%}" ><i class="fa fa-retweet "></i> Paired order</a>
        <a href="/orders/timebasedorder/" class="list-group-item {%
            if 'orders/timebasedorder/' in request.path%} active {%
            endif%}" ><i class="fa fa-clock-o "></i> Time based
            order</a>
        <a href="/orders/stoplossorder/" class="list-group-item {%
            if 'orders/stoplossorder/' in request.path%} active {%
            endif%}" ><i class="fa fa-ban "></i> Stop loss order</a>
      </div>
    </div>
    <div class="grid_3">
      <div class="row">
        {% block funds %}
          {% if funds %}
          <h4>Available funds:</h4><br>
            <table class="table table-hover table-bordered"
                style="width: auto;" summary="Funds in your wallet">
              <thead>
   <tr><th></th><th>Currency</th><th>Amount</th></tr>
              </thead>
              <tbody>
                {% for fund, amount in funds.items %}
    <tr><td><img src="{{ baseUrl }}/images/{{ fund}}_40x40.png"
        style="width:25px; height:25px;"></td><td><a
        href="/wallet/funds/{{ fund }}/">{{ fund }}</a></td><td>{{
        amount }}</td></tr>
                {% endfor %}
              </tbody>
            </table>
            {% else %}
              <p>No funds availables!</p>
          {% endif %}
        {% endblock %}
      </div>
    </div>
  <div class="grid_6">
      <div class="row">
        {% block message %}{% endblock %}
      </div>
      <div class="row" style="margin-left: auto; margin-right:
          auto;">
        {% block order %}{% endblock %}
      </div>
    </div>
  </div>
  <div class='row'>
    <div class='grid_3'></div>
    <div class="grid_9">
      {% block help %}{% endblock %}
    </div>
  </div><!-- end row-->
 </section>

{% endblock %}
```

Active orders template

Listing 10.33: Active orders template.

```
{% extends 'base.html' %}
{% block body_main %}
 {% load staticfiles %}
 {% load static %}
 {% static "" as baseUrl %}
 <section id="content" style="margin-bottom:60px; width:100%"
     class="wide-content">
   <div class="row">
     <div class="grid_3"><!-- aside menu content -->
       <div class="mini-submenu">
         <span class="icon-bar"></span>
         <span class="icon-bar"></span>
```

```
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </div>
        <div class="list-group" style="max-width:86%">
          <a  href="/orders/activeorders/" class="list-group-item {%
              if 'orders/activeorders/' in request.path%} active {%
              endif%}" ><i class="fa fa-play"></i> Active orders</a>
          <a href="/orders/simpleorder/" class="list-group-item {% if
              'orders/simpleorder/' in request.path%} active {%
              endif%}" ><i class="fa fa-circle-o"></i> Simple
              order</a></a>
          <a href="/orders/slicedorder/" class="list-group-item {% if
              'orders/slicedorder/' in request.path%} active {%
              endif%}" ><i class="fa fa-signal"></i> Sliced order</a>
          <a href="/orders/pairedorder/" class="list-group-item {% if
              'orders/pairedorder/' in request.path%} active {%
              endif%}" ><i class="fa fa-retweet "></i> Paired order</a>
          <a href="/orders/timebasedorder/" class="list-group-item {%
              if 'orders/timebasedorder/' in request.path%} active {%
              endif%}" ><i class="fa fa-clock-o "></i> Time based
              order</a>
          <a href="/orders/stoplossorder/" class="list-group-item {%
              if 'orders/stoplossorder/' in request.path%} active {%
              endif%}" ><i class="fa fa-ban "></i> Stop loss order</a>
        </div>
      </div>
      <div class="grid_9">
        <div class="row">
    {% if simple %}
      <h4>Simple orders:</h4>
            <table class="table table-bordered" data-height="400"
                style="width: auto;" summary="Simple active orders">
        <thead>
               <tr><th class="col-md-2">Id</th><th
                   class="col-md-2">Pair</th><th
                   class="col-md-2">Amount</th><th
                   class="col-md-2">Price</th><th
                   class="col-md-1">Total</th><th
                   class="col-md-1"></th></tr>
        </thead>
        <tbody>
          {% for k, v in simple.items %}
      {% ifequal v.type 'buy' %}
                   <tr style="background-color: rgba(0, 255, 0,
                       0.22);" ><td>{{ k }}</td><td>{{ v.pair
                       }}</td><td>{{ v.amount }}</td><td>{{ v.price
                       }}</td><td>{{ v.total }}</td><td><a
```

```
                        href="/orders/cancelsimple/{{ k
                        }}/">cancel</a></td></tr>
        {% endifequal %}
        {% ifequal v.type 'sell' %}
                    <tr style="background-color: rgba(255, 0, 0,
                        0.22);"><td>{{ k }}</td><td>{{ v.pair
                        }}</td><td>{{ v.amount }}</td><td>{{ v.price
                        }}</td><td>{{ v.total }}</td><td><a
                        href="/orders/cancelsimple/{{ k
                        }}/">cancel</a></td></tr>
        {% endifequal %}
                {% endfor %}
          </tbody>
              </table>
        <h5>Note: Buy orders are displayed in green and sell orders in
            red.</h5><br>
      {% else %}
        <p><h4>No simple orders active.</h4></p>
      {% endif %}
      {% if sliced %}
        <h4>Sliced orders:</h4>
              <table class="table table-bordered" data-height="400"
                  style="width: auto;" summary="Simple active orders">
          <thead>
                <tr><th class="col-md-1">Id</th><th
                    class="col-md-1">Pair</th><th
                    class="col-md-1">Amount</th><th class="col-md-2">#
                    orders</th><th class="col-md-1">Lower</th><th
                    class="col-md-1">Upper</th><th
                    class="col-md-1">Total</th><th
                    class="col-md-1">Active</th></th><th
                    class="col-md-1">Executed</th><th
                    class="col-md-1"></th></tr>
          </thead>
          <tbody>
            {% for k, v in sliced.items %}
              {% ifequal v.type 'buy' %}
                <tr style="background-color: rgba(0, 255, 0, 0.22);"
                    ><td>{{ k }}</td><td>{{ v.pair }}</td><td>{{
                    v.amount }}</td><td>{{ v.number }}</td><td>{{
                    v.lower }}</td><td>{{ v.upper }}</td><td>{{
                    v.total }}</td><td>{{ v.active }}</td><td>{{
                    v.executed }}</td><td><a
                    href="/orders/cancelsliced/{{ k
                    }}/">cancel</a></td></tr>
              {% endifequal %}
              {% ifequal v.type 'sell' %}
                        <tr style="background-color: rgba(255, 0, 0,
                            0.22);" ><td>{{ k }}</td><td>{{ v.pair
```

```
                              }}</td><td>{{ v.amount }}</td><td>{{
                              v.number }}</td><td>{{ v.lower
                              }}</td><td>{{ v.upper }}</td><td>{{ v.total
                              }}</td><td>{{ v.active }}</td><td>{{
                              v.executed }}</td><td><a
                              href="/orders/cancelsliced/{{ k
                              }}/">cancel</a></td></tr>
                 {% endifequal %}
                   {% endfor %}
           </tbody>
               </table>
         <h5>Note: If you cancel a a sliced order, all the single orders
             composing the sliced order will be canceled.</h5>
         <h5>Note: Buy orders are displayed in green and sell orders in
             red.</h5><br>
       {% else %}
         <p><h4>No sliced orders active.</h4></p>
       {% endif %}
       {% if paired %}
         <h4>Paired orders:</h4>
               <table class="table table-bordered" data-height="400"
                   style="width: auto;" summary="Paired active orders">
           <thead>
                   <tr><th class="col-md-2">Id</th><th
                       class="col-md-2">Pair</th><th
                       class="col-md-2">Amount</th><th
                       class="col-md-2">Price</th><th
                       class="col-md-1">Total</th><th
                       class="col-md-1">Contraprice</th><th
                       class="col-md-1"></th></tr>
           </thead>
           <tbody>
             {% for k, v in paired.items %}
         {% ifequal v.type 'buy' %}
                       <tr style="background-color: rgba(0, 255, 0,
                           0.22);" ><td>{{ k }}</td><td>{{ v.pair
                           }}</td><td>{{ v.amount }}</td><td>{{ v.price
                           }}</td><td>{{ v.total }}</td><td>{{ v.cprice
                           }}</td><td><a href="/orders/cancelpaired/{{ k
                           }}/">cancel</a></td></tr>
         {% endifequal %}
         {% ifequal v.type 'sell' %}
                       <tr style="background-color: rgba(255, 0, 0,
                           0.22);"><td>{{ k }}</td><td>{{ v.pair
                           }}</td><td>{{ v.amount }}</td><td>{{ v.price
                           }}</td><td>{{ v.total }}</td><td>{{ v.cprice
                           }}</td><td><a href="/orders/cancelpaired/{{ k
                           }}/">cancel</a></td></tr>
         {% endifequal %}
```

```
                {% endfor %}
        </tbody>
            </table>
      <h5>Note: Buy orders are displayed in green and sell orders in
          red.</h5><br>
    {% else %}
      <p><h4>No paired orders active.</h4></p>
    {% endif %}
    {% if time %}
            <table class="table table-bordered" data-height="400"
                style="width: auto;" summary="Time based active
                orders">
        <thead>
                <tr><th class="col-md-1">Id</th><th
                    class="col-md-1">Pair</th><th
                    class="col-md-1">Amount</th><th
                    class="col-md-1">Price</th><th
                    class="col-md-1">Total</th><th
                    class="col-md-2">Expiration time</th><th
                    class="col-md-1"></th></tr>
        </thead>
        <tbody>
          {% for k, v in time.items %}
      {% ifequal v.type 'buy' %}
                    <tr style="background-color: rgba(0, 255, 0,
                        0.22);" ><td>{{ k }}</td><td>{{ v.pair
                        }}</td><td>{{ v.amount }}</td><td>{{ v.price
                        }}</td><td>{{ v.total }}</td><td>{{ v.exp
                        }}</td><td><a href="/orders/canceltimebased/{{
                        k }}/">cancel</a></td></tr>
      {% endifequal %}
      {% ifequal v.type 'sell' %}
                    <tr style="background-color: rgba(255, 0, 0,
                        0.22);"><td>{{ k }}</td><td>{{ v.pair
                        }}</td><td>{{ v.amount }}</td><td>{{ v.price
                        }}</td><td>{{ v.total }}</td><td>{{ v.exp
                        }}</td><td><a href="/orders/canceltimebased/{{
                        k }}/">cancel</a></td></tr>
      {% endifequal %}
                {% endfor %}
        </tbody>
            </table>
      <h5>Note: Buy orders are displayed in green and sell orders in
          red.</h5><br>
    {% else %}
      <p><h4>No time based orders active.</h4></p>
    {% endif %}
    {% if stoplossorders %}
    {% else %}
```

```
      <p><h4>No stop loss orders active.</h4></p>
    {% endif %}
        </div>
      </div>
    </div><!-- end row-->
  </section>

{% endblock %}
```

Simple order template

Listing 10.34: Simple order template.

```
{% extends 'orders/orders_base.html' %}
{% block message %}
  {% if error %}
     <h4><font color="red"><b>{{ error }}</b></font></h4>
  {% elif success %}
     <h4><font color="success"><b>{{ success }}</b></font></h4>
  {% endif %}
{% endblock %}

{% block order %}
  <form action="" method="post">
    <h4>Create a simple order:</h4><br>
    {% csrf_token %}
    {{ form.non_field_errors }}
      <table>
        <tr><td>
    <p>
            {% for radio in form.pair %}
              {{radio}}
            {% endfor %}
    </p>
        </td></tr>
        <tr><td>
    <p>
            <font color="green"><b>{{form.buysell.0}}
                &nbsp&nbsp&nbsp&nbsp</b></font>
            <font color="red"><b>{{form.buysell.1}}</b></font>
    </p>
        </td></tr>
        <tr><td>
        <p>
          <div class="f_amount">
            <label for="id_amount">Amount: </label>
          </div>
      </td></tr>
  <tr><td>
          <div class="f_amount">
            {{ form.amount }}
          </div>
        </p>
        </td></tr>
        <tr><td>
        <p>
          <div class="f_price">
            <label for="id_price">Price: </label>
    </div>
        </td></tr>
  <tr><td>
          <div class="f_price">
            {{ form.price }}
    </div>
        </p>
        </td></tr>
        <tr><td>
    <p><input type="submit" value="Order" name="_order" class="btn
        btn-default"></p>
        </td></tr>
      </table>
  </form>
{% endblock %}

{% block help %}
  <p><h4>What is a simple order?</h4>
  A simple order allows you to create an order in the btc-e.com
      exchange.</p>
  <br>
  <p><h4>How to create it?</h4>
  After selecting the pair you want to trade with, choose the amount
      and the price to buy or sell the asset. Then press the order
      button to confirm.</p>

{% endblock %}
```

Sliced order template

Listing 10.35: Sliced order template.

```
{% extends 'orders/orders_base.html' %}

{% block message %}
  {% if error %}
```

```
     <h4><font color="red"><b>{{ error }}</b></font></h4>
  {% elif success %}
     <h4><font color="success"><b>{{ success }}</b></font></h4>
  {% endif %}
{% endblock %}

{% block order %}
  <form action="" method="post">
    <h4>Create a sliced order:</h4><br>
    {% csrf_token %}
    {{ form.non_field_errors }}
      <table>
        <td><tr>
      <p>
          {% for radio in form.pair %}
            {{radio}}
          {% endfor %}
      </p>
    </select>
        </tr></td>
        <td><tr>
          <font color="green"><b>{{form.buysell.0}}
              &nbsp&nbsp&nbsp&nbsp</b></font>
        </tr><tr>
          <font color="red"><b>{{form.buysell.1}}</b></font>
        </tr></td>
        <td><tr>
        <p>
          <div class="f_amount">
            <label for="id_amount">Amount: </label>
        </tr><tr>
            {{ form.amount }}
          </div>
        </p>
        </tr></td>
        <td><tr>
        <p>
          <td><tr>
          <div class="f_norders">
            <label for="id_norders">Number of orders: </label>
          </tr><tr>
            {{ form.numberOfOrders }}
          </div>
        </p>
        </tr></td>
        <p>
        <td><tr>
          <label for="id_lbound">Lower bound: </label>
          </tr><tr>
        {{ form.lowerBound }}
      </p>
      </tr></td>
      <p>
      <td><tr>
        <label for="id_ubound">Upper bound: </label>
        </tr><tr>
        {{ form.upperBound }}
      </p>
      </tr></td>
    <p><input type="submit" value="Order" name="_order" class="btn
        btn-default"></p>
        </tr></td>
      </table>
    </form>
{% endblock %}

{% block help %}
  <p><h4>What is a sliced order?</h4>
  A sliced order allows you to create several orders for the same
      asset, slicing the total amount into smaller suborders inside
      the specified range of prices.</p>
  <br>
  <p><h4>How to create it?</h4>
  After selecting the pair you want to trade with, choose the amount
      you want to splice in smaller orders. Then set the price
      boundaries in which the orders will be splitted. Finally press
      the order button to confirm.</p>
{% endblock %}
```

Time based order template

Listing 10.36: Time based order template.

```
{% extends 'orders/orders_base.html' %}
{% block header %}
  {% if form %} {{ form.media }} {% endif %}
{% endblock %}
{% block message %}
  {% if error %}
     <h4><font color="red"><b>{{ error }}</b></font></h4>
  {% elif success %}
     <h4><font color="success"><b>{{ success }}</b></font></h4>
  {% endif %}
{% endblock %}

{% block order %}
```

```
<form action="" method="post">
  <h4>Create a time based order:</h4><br>
  {% csrf_token %}
  {{ form.non_field_errors }}
    <table>
      <td><tr>
    <p>
        {% for radio in form.pair %}
          {{radio}}
        {% endfor %}
    </p>
  </select>
      </tr></td>
      <td><tr>
        <font color="green"><b>{{form.buysell.0}}
            &nbsp&nbsp&nbsp&nbsp</b></font>
      </tr><tr>
        <font color="red"><b>{{form.buysell.1}}</b></font>
      </tr></td>
      <td><tr>
      <p>
        <div class="f_amount">
          <label for="id_amount">Amount: </label>
      </tr><tr>
          {{ form.amount }}
        </div>
      </p>
      </tr></td>
      <td><tr>
      <p>
        <label for="id_price">Price: </label>
      </tr><tr>
        {{ form.price }}
      </p>
      </tr></td>
      <p>
      <td><tr>
        <div class="f_exptime">
          </tr><tr>
            <label for="id_exptime">Expiration time: </label>
            </tr><tr>
            {{ form.expiration_time }}
        </div>
      </p>
      </tr></td>
      <td><tr>
  <p><input type="submit" value="Order" name="_order" class="btn
      btn-default"></p>
      </tr></td>
    </table>
  </form>
{% endblock %}

{% block help %}
  <p><h4>What is a time based order?</h4>
  A time based order allows you to create an order that will expire
      in the selected time if it was not executed.</p>
  <br>
  <p><h4>How to create it?</h4>
  After selecting the pair you want to trade with, choose the amount,
      the price to buy or sell the asset and select the expiration
      date and time. Then press the order button to confirm.</p>
{% endblock %}
```

Paired order template

Listing 10.37: Paired order template.

```
{% extends 'orders/orders_base.html' %}
{% block message %}
  {% if error %}
     <h4><font color="red"><b>{{ error }}</b></font></h4>
  {% elif success %}
     <h4><font color="success"><b>{{ success }}</b></font></h4>
  {% endif %}
{% endblock %}

{% block order %}
  <table>
    <tr><td>
    <form action="" method="post">
      <h4>Create a paired order:</h4><br>
      {% csrf_token %}
      {{ orderForm.non_field_errors }}
    <table>
      <tr><td>
      <p>
         {% for radio in orderForm.pair %}
           {{radio}}
         {% endfor %}
        </p>
      </td></tr>
      <tr><td>
          <font color="green"><b>{{orderForm.buysell.0}}
              &nbsp&nbsp&nbsp&nbsp</b></font>
          <font color="red"><b>{{orderForm.buysell.1}}</b></font>
    </td></tr>
      <tr><td>
    <p>
        <div class="f_amount">
          <label for="id_amount">Amount: </label>
      </td></tr>
    <tr><td>
          {{ orderForm.amount }}
        </div>
    </p>
      </td></tr>
    <tr><td>
    <p>
        <div class="f_price">
          <label for="id_price">Price: </label>
    </td></tr>
  <tr><td>
          {{ orderForm.price }}
      </div>
      </p>
    </td></tr>
    <tr><td>
  <p><input type="submit" value="Order" name="_order" class="btn
      btn-default"></p>
      </td></tr>
     </form>
    </table>
</td>
<td style="vertical-align:top; padding:0 30px">
<table>
<form action="" method="post">
  <h4>Create a contra order:</h4><br>
  {% csrf_token %}
  {{ contraOrderForm.non_field_errors }}
      <tr><td>
  <p>
          {% for radio in contraOrderForm.opened %}
            {{radio}}
          {% endfor %}
  </p>
      </td></tr>
      <tr><td>
      <p>
        <div class="f_contraprice">
          <label for="id_contrprice">Contra order price: </label>
        </div>
      </td></tr>
<tr><td>
        <div class="f_contraprice">
```

```
          {{ contraOrderForm.contraprice }}
        </div>
      </p>
      </td></tr>

      <tr><td>
    <p><input type="submit" value="Contra order" name="_contraOrder"
        class="btn btn-default"></p>
      </td></tr>
  </form>
  </table>
  </td></tr>
  </table>
{% endblock %}

{% block help %}
  <p><h4>What is a paired order?</h4>
  A paired order is a composition of two orders. The first order is a
      normal order and the second one will have the opposite
      position, trading with the same amount. This kind of order
      allows you to view the benefit obtained when buying and selling
      or selling and buying an asset.
  <br><br>
  <p><h4>How to create it?</h4>
  For creating a paired order you have to fill the first form. Once
      the order is executed, you can select the id in the second
      form, and choose the price to trade with.
{% endblock %}
```

Stop loss order template

Listing 10.38: Stop loss order template.

```
{% extends 'orders/orders_base.html' %}
{% block message %}
  {% if error %}
     <h4><font color="red"><b>{{ error }}</b></font></h4>
  {% elif success %}
     <h4><font color="success"><b>{{ success }}</b></font></h4>
  {% endif %}
{% endblock %}

{% block order %}
  <form action="" method="post">
    <h4>Create a stop loss order:</h4><br>
    {% csrf_token %}
    {{ form.non_field_errors }}
    <table>
      <tr><td>
      <p>
          {% for radio in form.pair %}
            {{radio}}
          {% endfor %}
      </p>
      </td></tr>
      <tr><td>
      <p>
          <font color="green"><b>{{form.buysell.0}}
              &nbsp&nbsp&nbsp&nbsp</b></font>
          <font color="red"><b>{{form.buysell.1}}</b></font>
      </p>
      </td></tr>
      <tr><td>
        <p>
          <div class="f_amount">
            <label for="id_amount">Amount: </label>
          </div>
          </td></tr>
        <tr><td>
          <div class="f_amount">
            {{ form.amount }}
          </div>
      </p>
      </td></tr>
      <tr><td>
      <p>
        <div class="f_price">
          <label for="id_price">Threshold: </label>
      </div>
        </td></tr>
      <tr><td>
          <div class="f_price">
            {{ form.price }}
        </div>
        </p>
      </td></tr>
      <tr><td>
        </td></tr>
      <tr><td>
  <p><input type="submit" value="Order" name="_order" class="btn
      btn-default"></p>
      </td></tr>
    </table>
  </form>
{% endblock %}

{% block help %}
  <p><h4>What is a stop loss order?</h4>
  A stop loss order allows you to create an order which will be
      trigered in case that the market cross a defined threshold.
  <br><br>
  <p><h4>How to create it?</h4>
  Fill the form choosing the threshold price for which, if the market
      cross it, the order to trade will be created into the market.
  </p>

{% endblock %}
```

URLs

Listing 10.39: Order URLs.

```
from django.conf.urls import patterns, url, include

import views

urlpatterns = patterns('',
                    url(r'^activeorders/$', views.activeOrders,
                        name='activeOrders'),
                    url(r'^cancelsimple/(?P<order_id>\d+)/$',
                        views.cancelSimpleOrder,
                        name='cancelSimple'),
                    url(r'^cancelsliced/(?P<order_id>\d+)/$',
                        views.cancelSlicedOrder,
                        name='cancelSliced'),
                    url(r'^cancelpaired/(?P<order_id>\d+)/$',
                        views.cancelPairedOrder,
                        name='cancelPaired'),
                    url(r'^canceltimebased/(?P<order_id>\d+)/$',
                        views.cancelTimeBasedOrder,
                        name='cancelTimeBased'),
                    url(r'^simpleorder/$',
                        views.simpleOrder,name='simpleorder'),
                    url(r'^slicedorder/$', views.slicedOrder,
                        name='slicedOrder'),
                    url(r'^timebasedorder/$', views.timeBasedOrder,
                        name='timebasedorder'),
                    url(r'^pairedorder/$', views.pairedOrder,
                        name='pairedOrder'),
                    url(r'^stoplossorder/$', views.stopLossOrder,
                        name='stoplossOrder'),
                    )
```

10.4 Back end

10.4.1 Time order manager

Listing 10.40: Time based order manager.

```
#!/usr/bin/env python
import pika
import pickle
from datetime import datetime
import time
import logging
from os import environ
from sys import path
from pytz import timezone

def check_canceled_order(order):
    '''
    Return true if the status of the order is 'canceled', false
        otherwise
    '''
    order = TimeBasedOrder.objects.filter(btceid = order.btceid)[0]
    if order.status == 'canceled':
        loggger.info('Time backend: The order %s was canceled by a\
                user' % order.btceid )
        return True
    else:
        return False

def check_expired_order(order):
    '''
    Return true if the time_expiration_period has arrived, false
        otherwise
    '''
    if order.expiration_time <= datetime.now(timezone('UTC')):
        return True
    else:
        return False

def cancel_order(order):
    ''' Cancels an order'''
    # call btce api
    sk = order.user.userprofile.btce_secret_key
    ak = order.user.userprofile.btce_key
    # The order has to taken from the db to reflect the change in
        the status
    order = TimeBasedOrder.objects.filter(btceid = order.btceid)
    if len(order) == 0:
        logger.error('Something weird happend: The time queue
            contains a non\
        existing order')
        pass
    else:
        order = order[0]
        if order.status == 'canceled':
            logger.info('Expired a canceled order. Non doing nothing')
            pass
        else:
            oid = order.btceid
            r = cancel_order_btce(sk, ak, oid)
            order.status = 'expired'
            order.save()
            logger.info('Canceling order with id %s' % order.btceid)

def callback(ch, method, properties, body):
    '''
    If the order have expired: Mark it as expired.
    If the order has been canceled by the user: Don't add it to the
        queue
    Otherwise: Add it to the queue again
    '''
    order = pickle.loads(body)
    ch.basic_ack(delivery_tag = method.delivery_tag)
    logger.debug('Time backend: new order recieved with id %s' %
        order.btceid)
    if check_expired_order(order):
        cancel_order(order)
    elif check_canceled_order(order):
        pass
    else:
        ch.basic_publish(exchange = '', routing_key = 'order_timer',
            body = pickle.dumps(order), properties =
            pika.BasicProperties(delivery_mode = 2, ))
    time.sleep(10)

def main():
    chan_consumer.start_consuming()

if __name__ == '__main__':
    ''' Setup django environment '''
    environ['DJANGO_SETTINGS_MODULE'] = "MillonesApp.settings"
    path.append('/var/cryptomoneymakers/venv/MillonesApp/')
    from orders.models import TimeBasedOrder
    from wallet.btceapi import cancel_order as cancel_order_btce
    import django.db
    logger = logging.getLogger('wallet')

```

```
logger.debug('timer manager started')
con = pika.BlockingConnection(pika.ConnectionParameters(host =
    'localhost'))
chan_consumer = con.channel()
chan_consumer.basic_qos(prefetch_count=1)
chan_consumer.queue_declare(queue = 'order_timer', durable = True)
chan_consumer.basic_consume(callback, queue = 'order_timer')
main()
```

10.4.2 Tickers fetcher

Listing 10.41: Fetch tickers process.

```
#!/usr/bin/env python
import pika
from time import sleep
import json
import os
import sys

'''
This program query the btc-e API to get the tickers for all the pairs
and write them in the ticker fanout queue
'''

if __name__ == '__main__':
    # Setup environ
    os.environ['DJANGO_SETTINGS_MODULE'] = "MillonesApp.settings"
    sys.path.append('/var/cryptomoneymakers/venv/MillonesApp/')
    from btceapi import get_tickers
    from wallet.models import Change
    logger = logging.getLogger('wallet')

con =
    pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = con.channel()
channel.exchange_declare(exchange = 'tickers', type = 'fanout')

while True:
    try:
        t = Change.objects.values_list('label').all()
        tickers = get_tickers(t)
        tickerstr = json.dumps(tickers)
        channel.basic_publish(exchange='tickers', routing_key = '',
            body = tickerstr)
        sleep(2)
    except:
        logger.error('An exception ocurred while getting tickers for
            the fanout')
        pass

con.close()
```

10.4.3 Funds fetcher

Listing 10.42: Fetch user funds.

```
from django.core.management.base import BaseCommand, CommandError
from django.contrib.auth.models import User
from wallet.models import Funds
from wallet.btceapi import get_funds, calculate_total_usd

class Command(BaseCommand):
    args = '<username1 username2 ...>'
    help = 'Write into the database the current state of the users
        wallet'

    def handle(self, *args, **options):
        for uname in args:
            try:
                u = User.objects.get(username = uname)
                up = u.userprofile
            except:
                raise CommandError('User "%s" does not exist' % uname)

            funds = get_funds(str(up.btce_secret_key),
                str(up.btce_key), availables=False)
            totalusd = calculate_total_usd(funds=funds)

            #First add the total, and after add the currencies one by
                one
            f = Funds(currency = 'ttl', amount = totalusd, user = u)
            f.save()

            for fund in funds:
                f = Funds(currency = fund, amount = funds[fund], user
                    = u)
                f.save()
```

10.4.4 Feed executed orders

Listing 10.43: Feed executed orders.

```python
#!/usr/bin/env python2.7

'''
This script marks orders in the database as executed
if they had been executed.
It compares the result from the orders marked as
'active' in the database with the ones coming from the btce-e
API. It marks as a executed the one which are not in the second
list but in the first.
'''
import sys
import logging
from os import environ

if __name__ == '__main__':
    environ['DJANGO_SETTINGS_MODULE'] = "MillonesApp.settings"
    sys.path.append('/var/cryptomoneymakers/venv/MillonesApp/')
    from orders.models import *
    from wallet.btceapi import get_active_orders
    from django.contrib.auth.models import User
    import django.db
    logger = logging.getLogger('wallet')

def mark_executed_orders(btce_active, app_active):
    '''
    Marked the orders in app_active and not included in btce_active
        as executed
    '''
    for o in btce_active['return'].keys():
        # excluding order active in btc-e from the one in db
        #executed_orders = executed_orders.exclude(btceid=o)
        aod = app_active.exclude(btceid=o)

    # Mark the orders as 'executed' and save them into the db
    for o in app_active:
        if isinstance(o, PairedOrder):
            o.status = 'cont_executed'
        else:
            o.status = 'executed'
        o.save()
        logger.info('Marking order as executed: %s',o.id)

def mark_sliced_orders_as_finished(user):
    '''
    Marks sliced orders with all suborders executed as executed.
    '''
    asliced = SlicedOrder.objects.filter(user = user, status =
        'started')
    for sliced in asliced:
        finished = 1
        for simple in sliced.btceOrders.all():
            if simple.status == 'started':
                finished = 0
        if finished == 1:
            sliced.status = 'executed'
            sliced.save()
            logger.info('marked sliced order %s as a executed',
                sliced_order.id)


# TODO removed hard coded username
user = User.objects.all()[2]
sk = user.userprofile.btce_secret_key
ak = user.userprofile.btce_key
executed_orders=[]
# get oders marked as a 'active' on the database
aod = BtceOrder.objects.filter(user = user).filter(status='started')
# get orders marked as a active in btc-e
aob = get_active_orders(sk, ak)
# get list of orders which are in database but not in btc-e
if aob == None or 'error' in aob['return']:
    # This is not an error if the user has no active orders at the
        moment.
    # TODO solve issue
    logger.error('Error getting the list of active orders for user
        %s',user.username)
    #sys.exit(-1)
    pass
else:
    for o in aob['return'].keys():
            # excluding order active in btc-e from the one in db
            #executed_orders = executed_orders.exclude(btceid=o)
            aod = aod.exclude(btceid=o)

    # Mark the orders as 'executed' and save them into the db
    for o in aod:
        o.status = 'executed'
        o.save()
        logger.info('Marking order as executed: %s',o.id)

# Mark also the ones for pairedorder
aod = PairedOrder.objects.filter(user =
    user).filter(status='cont_started')
if len(aod)> 0:
```

```python
    mark_executed_orders(aob, aod)

# Now, that the simple/btce orders are marked as executed, let's see if some
# sliced order is completed.
mark_sliced_orders_as_finished(user)
```

10.4.5 Store tickers

Listing 10.44: Fetch tickers.

```python
from django.core.management.base import BaseCommand, CommandError
from wallet.models import Ticker
from wallet.btceapi import get_tickers

'''
TODO: This functions has to be deprecated in favour on a process that reads
from the tickers fanout queue and write the results into the database
'''

class Command(BaseCommand):
    args = 'label1 label2 ...'
    help = 'It fetchs the current value of a ticket and writes into the db'

    def handle(self, *args, **options):
        for lab in args:
            try:
                t = get_tickers([lab])
            except:
                raise CommandError('Unable to retrieve the ticker for %s' %lab)

            t = t[lab]
            f = Ticker(label = lab, high = t['high'], low = t['low'], avg = t['avg'],
                    vol = t['vol'], vol_cur = t['vol_cur'], last = t['last'], buy = t['buy'],
                    sell = t['sell'], updated = t['updated'], server_time = t['server_time'])
            f.save()
            self.stdout.write('Wroted ticker for %s' % lab)
```

10.4.6 btc-e API methods

Listing 10.45: btc-e API methods.

```python
from __future__ import division
import urllib
import json
import hmac
import httplib
import hashlib
from utils.utils import get_nonce

def remove_key(d,key):
   r = dict(d)
   del r[key]
   return r


'''Public btc-e api'''
def get_tickers(labels=None):
    '''
    This function receives a list of valid btc-e labels and return a
        dictionary composed by label:ticker
    :param labels: list of labels to retrieve the tickers
    :return: Dictionary label:ticker
    '''
    if labels == None:
        from wallet.models import Change
        labels = Change.objects.all()
        labels = [ p.label for p in labels ]

    tickers={}
    #TODO: create threads for paralel execution
    for label in labels:
        conn=httplib.HTTPSConnection('btc-e.com')
        url = '/api/2/'+ label + '/ticker/'
        conn.request("GET",url)
        response = conn.getresponse()
        respjson=json.load(response)
        try:
            tickers[label] = respjson['ticker']
        except:
            tickers[label] = respjson['error']
            # TODO logging the error
        conn.close()

    return tickers

def get_trades(pair):
    con = httplib.HTTPSConnection('btc-e.com')
    url = '/api/2/'+pair + '/trades'
    con.request('GET',url)
    resp= con.getresponse()
    respjson=json.load(resp)
    con.close()
    return respjson

def get_depth(pair,number=None):
    conn=httplib.HTTPSConnection('btc-e.com')
    if number:
        conn.request("GET",'/api/2/'+pair+'/depth/'+number)
    else:
        conn.request("GET",'/api/2/'+pair+'/depth')
    response = conn.getresponse()
    respjson=json.load(response)
    conn.close()
    return respjson

'''Private btc-e api'''
def get_info(sk, ak):
    '''This function call the getInfo method from the btc-e api
    retrieving a python dict with the next info:
     -Funds
     -Rights of current API key
     -Transaction count
     -Open orders
     -Server time'''
    params = {"method":"getInfo","nonce": get_nonce()}
    params = urllib.urlencode(params)
    # Hash the params string to produce the Sign header value
    H = hmac.new(sk, digestmod=hashlib.sha512)
    H.update(params)
    sign = H.hexdigest()

    headers = {"Content-type": "application/x-www-form-urlencoded",
                     "Key": ak,
                     "Sign": sign}
    conn = httplib.HTTPSConnection("btc-e.com")
    conn.request("POST", "/tapi", params, headers)
    response = conn.getresponse()
    respjson=json.load(response)
    if respjson.has_key('error'):
        # logging error
        print(respjson['error'])
        conn.close()
        return
    info = respjson
    conn.close()
    return info


def get_active_orders(secretkey,apikey):
    '''
    Query the btc-e API to get the list of active orders for a user
    '''
    #mehod name and nonce go into the POST parameters
    params = {"method":"ActiveOrders","nonce": get_nonce()}
    params = urllib.urlencode(params)
    # Hash the params string to produce the Sign header value
    H = hmac.new(secretkey, digestmod=hashlib.sha512)
    H.update(params)
    sign = H.hexdigest()

    headers = {"Content-type": "application/x-www-form-urlencoded",
                        "Key": apikey,
                        "Sign": sign}
    conn = httplib.HTTPSConnection("btc-e.com")
    conn.request("POST", "/tapi", params, headers)
    response = conn.getresponse()
    respjson=json.load(response)
    if respjson.has_key('error'):
        print(respjson['error'])
        conn.close()
        return

    activeorders=respjson
    conn.close()
    return activeorders


def get_funds(secretkey, apikey, nround=0, availables=True,
    null=False):
    '''
    Get the funds from a user
        if nround>0 return the funds rounded to nround decimals
        if availables = True, get only the availables, if Flase, get
            also the one
        in get_active_orders
    '''
    params = {"method":"getInfo","nonce": get_nonce()}
    params = urllib.urlencode(params)
    h = hmac.new(secretkey, digestmod=hashlib.sha512)
    h.update(params)
    sign = h.hexdigest()
    headers = {"Content-type": "application/x-www-form-urlencoded",
                        "Key": apikey,
                        "Sign": sign}
    conn = httplib.HTTPSConnection("btc-e.com")
    conn.request("POST", "/tapi", params, headers)
    response = conn.getresponse()
    respjson=json.load(response)
    conn.close()

    if respjson.has_key('error'):
        return respjson['error']

    funds=respjson['return']['funds']


    # Now sum the funds which are in active orders
    if availables == False:
        ao = get_active_orders(secretkey,apikey)
        if ao != None and ao['success'] == 1:
            ao = ao['return']
            for o in ao:
                p = ao[o]['pair']
                p = p.split('_')
                if ao[o]['type'] == 'sell': #if buy, the currency
                    owned is the first in the pair
                    cur=p[0]
                    funds[cur] += ao[o]['amount']
                else:
                    cur=p[1]
                    funds[cur] += ao[o]['amount']

    if nround>0:
        for f in funds:
            funds[f]=round(funds[f],nround)
    # eliminate null values
    if null == False:
        funds = {k:v for k, v in funds.iteritems() if v>0}

    return funds

def create_order(secretkey,apikey,pair,sellbuy,rate,amount):
    params = {"method":"Trade","nonce":
        get_nonce(),"pair":pair,"type":sellbuy,"rate":rate,"amount":amount}
    params = urllib.urlencode(params)
    H = hmac.new(secretkey, digestmod=hashlib.sha512)
    H.update(params)
    sign = H.hexdigest()

    headers = {"Content-type": "application/x-www-form-urlencoded",
                    "Key":apikey,
                    "Sign":sign}
    conn = httplib.HTTPSConnection("btc-e.com")
    conn.request("POST", "/tapi", params, headers)
    response = conn.getresponse()
    respjson=json.load(response)
    conn.close()
    #if respjson['error']:
    #    logger.error("btc-eAPI: New order: %s" % respjson['error'])
    #else
    #    logger.info('btc-eAPI: New order created correctly')
    return respjson


def cancel_order(secretkey,apikey,orderid):
    params = {"method":"CancelOrder","nonce":
        get_nonce(),"order_id":orderid}
    params = urllib.urlencode(params)

    # Hash the params string to produce the Sign header value
    H = hmac.new(secretkey, digestmod=hashlib.sha512)
    H.update(params)
    sign = H.hexdigest()

    headers = {"Content-type": "application/x-www-form-urlencoded",
                        "Key":apikey,
                        "Sign":sign}
    conn = httplib.HTTPSConnection("btc-e.com")
    conn.request("POST", "/tapi", params, headers)
    response = conn.getresponse()
    respjson=json.load(response)
    if respjson.has_key('error'):
        return respjson['error']

    res=respjson['return']
    conn.close()
    return res

def
    calculate_total_usd(secretkey=None,apikey=None,funds=None,availables=True):
    #TODO get from db all the available tickers/labels and pass to
        get_tickers
    labels=['btc_usd', 'ltc_usd', 'nmc_usd', 'ppc_usd', 'eur_usd',
        'nvc_usd', 'usd_rur']
    changes={'btc':'btc_usd', 'ltc':'ltc_usd', 'nmc':'nmc_usd',
            'nvc':'nvc_usd','ppc':'ppc_usd', 'eur':'eur_usd',
                'rur':'usd_rur'}
    tickers = get_tickers(labels)
    total = 0
    if funds == None:
        if availables == True:
            funds = get_funds(secretkey,apikey,availables=True)
        else:
            funds = get_funds(secretkey,apikey,availables=False)

    for c in funds:
        if funds[c] > 0.0:
            if 'usd' in c:
                total = total + funds[c]
            else:
                change = changes[c]
                if 'usd' in change:
                    # This means there is a direct path from 'currency'
                        to usd
                    usd = tickers[change]['last'] * funds[c]
                    total = total + usd
                else:
                    # This happens when is not direct change between
                        currency and usd,
                    # change first to btc
                    btc = tickers[change]['last'] * funds[c]
                    usd = tickers['btc_usd']['last'] * btc
                    total = total + usd
    return total

#TODO change params for the parameters
def trade_history(secretkey,apikey,params):
    '''Retrieve history of trades from btc-e
    Atributes:

    secretkey -- User's btce secret key
    apikey -- User's btce secret api
    params -- #TODO fill me!
    '''
    params["method"]="TradeHistory"
    params["nonce"]= get_nonce()

    params = urllib.urlencode(params)
    H = hmac.new(secretkey, digestmod=hashlib.sha512)
    H.update(params)
    sign = H.hexdigest()

    headers = {"Content-type": "application/x-www-form-urlencoded",
                     "Key":apikey,
                     "Sign":sign}
    conn = httplib.HTTPSConnection("btc-e.com")
    conn.request("POST", "/tapi", params, headers)
    response = conn.getresponse()
    respjson=json.load(response)
    if respjson.has_key('error'):
        return respjson['error']

    conn.close()
    return respjson['return']


def transaction_history(secretkey,apikey,params):
    '''Retrieve history of transfers from btc-e'''
    params["method"]="TransHistory"
    params["nonce"]= get_nonce()

    params = urllib.urlencode(params)
    H = hmac.new(secretkey, digestmod=hashlib.sha512)
    H.update(params)
    sign = H.hexdigest()

    headers = {"Content-type": "application/x-www-form-urlencoded",
                     "Key":apikey,
                     "Sign":sign}
    conn = httplib.HTTPSConnection("btc-e.com")
    conn.request("POST", "/tapi", params, headers)
    response = conn.getresponse()
    respjson=json.load(response)
    if respjson.has_key('error'):
        return respjson['error']

    conn.close()
    return respjson['return']
```