# BankChain - Architecture design

Chainable Technologies

H.G. van de Kuilen
rvandekuilen, 4226151
J.C. Kuijpers
jckuijpers, 4209915
M.R. Kok
mrkok, 4437659
I. Dijcks
idijcks, 4371151

May 2017

# Contents

# 1 Introduction

In this document, a high-level description of our app is given, together with the design goals. There is a large focus on the subsystem decomposition and how practical problems, like persistence and concurrency are solved. The app being created is an additive to a web-of-trust on a mobile phone. It adds the possibility to verify a peer using their public key and bank account, by sending a cent to the peer with a challenge. The solved challenge is sent back using another cent. The product relies on the difficulty to get a bank account in a false name or on illegal markets.

## 1.1 Design goals

The product being created is foremost a research project and a proof-of-concept. This is the reason the three main design goals are security and code quality.

### 1.1.1 Secure

As the app is about getting verification of a peer in a secure, encrypted and trusted way, making the app secure is an important part of the design. Encrypted challenges over bank accounts is all nice and good, unless your private key can easily be stolen by a simple backdoor in the app.

### 1.1.2 High code quality

This project being foremost a proof-of-concept of an extension to a web-of-trust, code quality is very important. With high code quality, the processes, advantages but also the limitations can be easily explained to future developers (of any project) and researchers. In general, higher code quality also improves the development velocity (because of less bugs and more understanding of the code), product quality (because of less bugs and more reviews) and mindset of the developers (less issues, more understanding, more freedom in code changes due to testing).

To get higher code quality, a lot of the code must be tested and code must be reviewed by peers. The program needs a clear structure, both for testability (mocking), to be able to switch components and to be able to work on different parts in parallel.

# 2 Software architecture views

In this section we describe what the architecture of the code and the product is.

## 2.1 Subsystem decomposition

The system is divisable in two main parts: a GUI and all functionality. The functionality can be divided in many parts.

### Bank interface

The connection to the bank is implemented using an set of classes that cover the API calls to the bank and handle all other bank related steps. This is an abstracted area, so multiple banks can be implemented. The main implementation is with the Bunq API. There is also a mock implementation of a bank so other code can be tested without actually calling into a bank. The Bunq implementation uses a HTTP library named Retrofit (Square, Inc., 2017).

### Cryptographic signing system

Our cryptographic signing system uses asymmetric key cryptography to sign and verify a string in the description of the bank transfers. We use the ed25519 cryptographic system for this task. Imagine the following scenario: Alice and Bob both have a keypair and a bank account. They wish to verify eachothers keypair and bank account. Using our software, Alice generates a random string and signs it with her signing key. She then sends a bank transfer of one cent to Bob, with in the description the random string and the signature. Bob then verifies the signature against Alice's verifying key. If the signature is valid, Bob knows that some person (presumably Alice) controls both that bank account, with Alice's name attached to it, and that signing key. Bob then sends back a bank transfer of one cent with the same random string, but this time with his own signature. Alice can then verify that some person (presumably Bob) controls both that bank account, with Bob's name attached to it, and that signing key. Now they both broadcast their new validated findings onto their own blockchains.

Our cryptography package strives to keep it's external API as simple as possible to hide fault-prone cryptographic code. The only methods exposed are responsible for generating a challenge, a response and checking for validity of those challenges and responses.

The bank interface also has a class that is connected to the cryptographic package, the transaction parser. This parser can respond to pending challenges, and retrieve the verified combinations of keypairs and bank accounts.

### Blockchain interface

As we don't implement a blockchain but still need to talk to it, we will have a dummy blockchain implementation that uses memory and files instead. The API of this will be cooperated with the project that write an actual blockchain. For this reason, there will be multiple implementations of the blockchain interface: a mock, a dummy and possibly an actual blockchain.

The blockchain mock is also capable of creating a public and private keypair for the user. The public key is stored in blockchain, and the private key is stored on the device.

#### Verification logic

This system contains the business logic of the verification. It receives requests from the API, asks for data from the blockchain, creates a challenge and sends it with the bank interface.

#### GUI

The graphical user interface shows information to the user and accepts input. The GUI is never talked to by other components: the GUI only talks to them. To receive notifications and updates, it will need to listen on the bus. See the concurrency chapter.

## 2.2 Hardware/software mapping

This system uses third party communication, the third party being a bank. The implemented bank is Bunq.

All software written in this project resides on two peers, A and B. Both are Android phones and have the BankChain app installed. They are the same, equal peers, both server and client. A peer can send a transaction to a bank using the bank implementation. The whole flow is handled by the Challenge/Response subsystem. It calls into the other subsystems using direct calls.

All actions are either started using timers, start-events or user interactions.

## 2.3 Persistent data management

There are a couple of persistent data items for BankChain: a list of previously verified IBANs, settings like the bank API code, a private key and previous verifications.

The verified IBANs would, when combined with the other project groups, be stored in the blockchain. The private key would be handled by group 1. The only things left is settings, and a list of verifications for display. This can be done using a file based storage.

As the app is not merged with other apps, at this point (or possibly at all), there is storage for a dummy blockchain and a private and public key pair. This is not in the scope of the project but is useful for testing and developing the application. Both will be stored in files.

The bank (Bunq) session is also persisted to the device, to decrease app loading times and improve stability of the keys. The private-public keypair for Bunq is stored in the Android KeyStore: a secure storage. The rest of the session is stored on local storage (not external storage like an SD card). This also contains the Bunq public key.

## 2.4 Concurrency

As the GUI must remain usable, even when running longer operations, long operations should not be run by the main thread. To do this, Android can use threads and runnables. Together with anonymous function, one can run code in the background. However, to do any GUI updates, there must be a switch to the main thread again (Google, Inc., 2017). This gets very messy, especially when going multiple levels deep. To solve this issue, we make use of a Java 8 feature called CompletableFutures. These allow functions to run in other threads and provide neat and clean code to work with their return values, wait for threads, combine results, and so on (*Java 8: Writing asynchronous code with CompletableFuture*, 2016). The code that runs inside the GUI can jump to the main thread using a utility function. Together with Java 8 lambdas, the code for this looks quite good.

# Glossary

**API** Application Programming Interface. 3, 4

**blockchain** is a distributed database that maintains a continuously growing list of records, called blocks, secured from tampering and revision (Wikipedia, 2017). 3, 4

**bunq** is a Dutch payment app and bank that provides a developer friendly and open programming interface.. 3

**GUI** Graphical User Interface. 4

**HTTP** HyperText Transfer Protocol. 3

# References


Google, Inc. (2017, March 21). *Processes and threads.* Retrieved May 1st, 2017, from `https://developer.android.com/guide/components/processes-and-threads.html`

*Java 8: Writing asynchronous code with completablefuture.* (2016, May 24). Retrieved May 12th, 2017, from `http://www.deadcoderising.com/java8-writing-asynchronous-code-with-completablefuture/`

Square, Inc. (2017). *Retrofit.* Retrieved May 1st, 2017, from `http://square.github.io/retrofit/`

Wikipedia. (2017). *Blockchain — wikipedia, the free encyclopedia.* Retrieved from `https://en.wikipedia.org/w/index.php?title=Blockchain&oldid=778628009` ([Online; accessed 4-May-2017])