Summer of Bitcoin 2023 Project Proposal

GARVIT JOSHI
Indian Institute of Technology(IIT-BHU), Varanasi
Varanasi, India

Summer of Bitcoin 2023

# Padawan Wallet

# Advanced Bitcoin features on Padawan Wallet


➢ **Personal Information** **3**
➢ **Synopsis** **4**
➢ **Project Plan** **4**
1- Typical advanced features to the wallet app that other mobile wallet perform, namely: 5
2- What is the BDK library and its features? 6
3- Adding advanced features to the wallet 7
3.1- Multi-signature support 7
3.2- Multi Currency Support 10
3.3- Advanced fee control [Replace-by-Fee (BIP125)] 12
3.4- Custom scripting 14
3.5- Transaction batching 15
3.6- Custom fee estimation 16
3.7- Send to Multiple recipients 17
3.8- Partial signatures on PSBTs 18
3.9- Send All functionality 20
3.10- Create OP_RETURN outputs 21
3.11- Choose custom Electrum server 22
4- Adding features to the faucet to build its resilience so that the stress on the faucet can be reduced. 24
4.1- Implement anti-bot measures 24
4.2- Implement withdrawal limits 26
4.3- Implement a queue system 28
4.4- Monitor and adjust 31
➢ **Linked Technologies** **32**
➢ **Project Timeline** **32**
➢ **Future Deliverables** **34**
➢ **Benefits to Community** **35**

➢ **Post SoB** **35**
➢ **Conclusion** **36**

# ➢ Personal Information

This is Garvit Joshi from Varanasi, India and I am a 2nd year student pursuing my B.Tech in Mining engineering from IIT BHU. I was introduced to the world of programming and software development in my first year. Since then, I have been very enthusiastic about deep diving into various fields of Computer Science. The areas that capture my interests are Data Structures and Algorithms, Operating Systems, App development, Web Development and computer vision. For most of my programming journey, I have worked primarily with Web-based technologies, my niche and C/C++ programs, and I have recently been diving into App Development as well.

➢ Github username: Garvit2003
➢ Linkedin: (23) Garvit Joshi | LinkedIn
➢ Email: garvit.joshi.min21@itbhu.ac.in
➢ Phone Number:(+91) 8126696009
➢ University: Indian Institute of Technology BHU (Varanasi)
➢ Time-zone: Indian Standard Time (GMT +5:30)
➢ Address: Varanasi, Uttar Pradesh - 221002, India
➢ Project Size: 350 hours (Large)

## ➢ Synopsis

The target of our project is to implement and add advanced features into our wallet that other mobile wallets perform. We also need to solve the issues of stress on the Tatooine faucet and to build its resilience.

## ➢ Project Plan

➢

The roadmap of our project basically includes:

1. Typical advanced features of the wallet app that other mobile wallets perform.
2. What is the BDK library and its features?
3. Adding advanced features to the wallet.
4. Adding features to the faucet to build its resilience so that the stress on the faucet can be reduced.

## ➢ Detailed Description of steps of Project Plan

### 1- Typical advanced features to the wallet app that other mobile wallet perform, namely:

1) Multi Signature Support
2) Multi Currency Support
3) Advanced fee control [Replace-by-Fee (BIP125)]
4) Custom scripting
5) Transaction batching
6) Custom fee estimation
7) Send to multiple recipients
8) Partial signatures on PSBTs
9) "Send All" functionality
10) Create OP_RETURN outputs
11) Choose custom Electrum server

We are going to add these features using the bdk library, so lets understand what the BDK library is.

## 2- What is the BDK library and its features?

The "bdk" library is a Bitcoin development kit for Rust, which provides a set of tools and utilities for building Bitcoin applications. Here are some of the things that the bdk library can do:

a) Create and manage Bitcoin wallets: The bdk library provides a simple and secure way to create, manage and store Bitcoin wallets. It supports multiple wallet types, including HD wallets, and allows users to generate and manage their own private keys.
b) Send and receive Bitcoin transactions: With the bdk library, we can easily send and receive Bitcoin transactions using standard Bitcoin network protocols. It also supports advanced transaction features such as multi-signature and time-locked transactions.
c) Query Bitcoin network data: The bdk library provides an easy way to query Bitcoin network data, including transaction history, balances, and unspent transaction outputs (UTXOs).
d) Support for multiple Bitcoin network protocols: The bdk library supports multiple Bitcoin network protocols, including Bitcoin Core, Electrum, and Blockstream Green.
e) Integration with other Bitcoin libraries: The bdk library is designed to be easily integrated with other Bitcoin libraries, such as the Bitcoin scripting library or the Bitcoin cryptography library.
f) Support for hardware wallets: The bdk library supports hardware wallets such as Ledger and Trezor, allowing us to manage our Bitcoin wallets securely on a hardware device.

Overall, the bdk library provides a comprehensive set of tools and utilities for developers looking to build Bitcoin applications in Rust.

## 3- Adding advanced features to the wallet

The bdk library provides a wide range of advanced features for building Bitcoin wallets. Here are a few of the most notable ones:

### 3.1- Multi-signature support

The bdk library provides built-in support for creating and managing multi-signature wallets, which require multiple parties to sign transactions before they can be broadcast to the Bitcoin network.

The steps to add multi-signature support to padawan wallet are-

a. Define a multi-signature policy: Define a policy that specifies the required signatures for a transaction. The bdk library provides a **policy::cosigner::Cosigner** struct that we can use to define our policy.
b. Create a new wallet instance with the policy: Create a new bdk wallet instance that uses the policy. We can do this by passing the policy to the **Wallet::new** function.
c. Integrating the multi-signature wallet with our existing wallet: Integrate the multi-signature wallet with our existing wallet's code by creating functions that call the necessary bdk functions to generate addresses, sign transactions, etc.

d. Sharing the addresses and policy with other signers: To use the multi-signature wallet, we need to share the generated addresses and policy with other signers so they can sign transactions.
e. Sign and broadcast transactions: Once we have collected the necessary signatures, we can sign and broadcast transactions as we would with a regular wallet.

Here is the code snippet for the steps above:

```rust
use bdk::bitcoin::secp256k1::Secp256k1;
use bdk::bitcoin::util::bip32::{ExtendedPrivKey, ExtendedPubKey};
use bdk::bitcoin::{Address, Network, Script, TxOut};
use bdk::database::MemoryDatabase;
use bdk::keys::{DerivableKey, ExtendedKey};
use bdk::miniscript::descriptor::{Descriptor, DescriptorMeta};
use bdk::wallet::{AddressIndex, AddressInfo, KeychainKind, OfflinePolicy, Wallet};
use std::str::FromStr;

fn main() {
    // Define the cosigners for the multi-signature wallet
    let cosigners = vec![
        (
            ExtendedPubKey::from_str(
                "tpubDCdGFhS6rxHfSTmxk1TrX8WgU1T6c7vpmpUaT6qzW6GMsD6iHQjKdGuRvXgaW
            )
            .unwrap(),
            None,
        ),
        (
            ExtendedPubKey::from_str(
                "tpubDCiNgjJwLWoQoUdnnf7hQECU6GKei6CqrU4Ge4U19Cahh1HwUe85GnSd5b8EM
            )
            .unwrap(),
            None,
        ),
        (
            ExtendedPubKey::from_str(
                "tpubDC7uATYJAd84tQ9TtTquT1GJux1fwAmNU7V3qW3G15eVZgPnygoJ7e9XWytjK
            )
            .unwrap(),
            None,
        ),
    ];
```

```rust
// Define the multi-signature policy
let policy = bdk::wallet::policy::multi::Multi::new(cosigners.clone(), 2).unwrap();

// Define the descriptor for the multi-signature wallet
let descriptor = Descriptor::<PublicKey>::new_multisig(&policy, None, None).unwrap();

// Create a new bdk wallet instance that uses the policy and descriptor
let wallet = Wallet::new(
    descriptor.clone(),
    None,
    Network::Testnet,
    MemoryDatabase::default(),
    None,
    OfflinePolicy::static_(),
    policy,
)
.unwrap();

// Define functions that call the necessary bdk functions to generate addresses, sign transactions, etc.
// Here is an example of a function to generate a multi-signature address:
fn generate_address(
    wallet: &Wallet<MemoryDatabase<bitcoin::util::psbt::PartiallySignedTransaction>, bdk::blockchain::EsploraBlockchain>,
    index: AddressIndex,
) -> Result<AddressInfo, bdk::Error> {
    wallet.get_address(index, None)
}

// Integrate the multi-signature wallet with your existing wallet's code
// Here is an example of a function
```

## 3.2- Multi Currency Support

Multi currency support enables users to experiment on different types of digital assests.

The following steps can be taken to enable multi-currency support:

1. Add support for multiple cryptocurrencies: In order to support multiple currencies, the wallet needs to be able to recognize and handle different types of cryptocurrencies. This can be achieved by adding support for the relevant cryptocurrency network protocols and APIs.

2. Store multiple sets of keys: Each cryptocurrency has its own set of private and public keys. In order to support multiple currencies, the wallet needs to be able to store and manage multiple sets of keys.
3. Add support for multi-currency transactions: Transactions involving multiple currencies require special handling. The wallet needs to be able to identify the currencies involved and convert between them as necessary.
4. Display balances and transaction histories for each currency: To enable users to manage their funds across multiple currencies, the wallet needs to display the relevant balances and transaction histories.

```
// define a class to represent a cryptocurrency
data class Cryptocurrency(val name: String, val symbol: String, val network: String)

// define a list of supported cryptocurrencies
val supportedCurrencies = listOf(
    Cryptocurrency("Bitcoin", "BTC", "bitcoin"),
    Cryptocurrency("Ethereum", "ETH", "ethereum")
)

// initialize the wallet with the selected cryptocurrency
fun initWallet(currency: Cryptocurrency) {
    // initialize the wallet with the selected cryptocurrency network
    // initialize the keys for the selected cryptocurrency
    // display the balance and transaction history for the selected cryptocurrency
}

// function to switch between cryptocurrencies
fun switchCurrency(currency: Cryptocurrency) {
    // save the current balance and transaction history
    // switch to the selected cryptocurrency
    // display the balance and transaction history for the selected cryptocurrency
}

// function to handle multi-currency transactions
fun sendMultiCurrency(amount: Double, fromCurrency: Cryptocurrency, toCurrency: Cryptocurrency) {
    // convert the amount from the fromCurrency to the toCurrency
    // create and sign the multi-currency transaction
    // broadcast the transaction to the relevant networks
    // update the balance and transaction history for each currency involved
}
```

## 3.3- Advanced fee control [Replace-by-Fee (BIP125)]

The bdk library provides fine-grained control over Bitcoin transaction fees, including the ability to set custom fee rates and specify transaction priorities.

Replace-by-fee (RBF) is a feature in the Bitcoin protocol that allows us to increase the fee of a transaction that has not yet been confirmed in the blockchain. This is achieved by creating a new transaction with a higher fee and a different transaction ID, effectively replacing the original transaction.

The purpose of RBF is to allow users to adjust their transaction fees in response to changing network conditions, such as high transaction volumes or sudden spikes in demand. To use Replace-by-fee (RBF) in the padawan wallet app that is built using the BDK library, we can follow these steps:

a. First, we need to ensure that the wallet app supports the BDK library and can be integrated with it.

b. Once the BDK library is integrated, we can start using the RBF feature in our wallet app by marking the transactions as replaceable. We can do this by setting the replaceable flag when creating a transaction using the **build_tx()** method of the Wallet struct.

```
let mut builder = wallet.build_tx();
builder.enable_rbf();
// Add inputs and outputs to the transaction
let (psbt, _) = builder.finish()?;
```

c. After marking a transaction as replaceable, we can use the **bump_fee()** method of the Wallet struct to create a new transaction with a higher fee that replaces the original transaction. This method takes the transaction ID of the original transaction as input and returns the ID of the new transaction.

```
let original_txid = "..."; // Replace with the ID of the original transaction
let mut bump_options = BumpFeeOptions::default();
bump_options.fee_rate = FeeRate::from_sat_per_vb(100); // Set the new fee rate
let new_txid = wallet.bump_fee(&original_txid, Some(bump_options))?;

```

d. Finally, we can broadcast the new transaction to the Bitcoin network using the broadcast() method of the Wallet struct.

```
let new_txid = "..."; // Replace with the ID of the new transaction
let tx = wallet.get_transaction(&new_txid)?;
wallet.broadcast(&tx)?;

```

This code retrieves the new transaction using its ID and broadcasts it to the Bitcoin network.

## 3.4- Custom scripting

Custom scripting in the BDK library refers to the ability to create and execute custom Bitcoin scripts using the library's scripting engine. This allows for more advanced transaction types, such as **multisig transactions** and complex smart contracts, to be implemented within a Bitcoin wallet or application.

The general steps involving enabling of custom scripting in our wallet are:

a. Add the BDK library to our project dependencies.
b. Initialize the BDK wallet object.
c. Create a custom script using the BDK scripting engine.
d. Build and sign a transaction using the custom script.

```
import org.bitcoindevkit.bdkjni.*
import org.bitcoindevkit.bdkjni.Types.*
import org.bitcoindevkit.bdkjni.Script.*

// Step 1: Add BDK library to project dependencies
dependencies {
    implementation("org.bitcoindevkit:bitcoindevkit:0.3.0")
}

// Step 2: Initialize BDK wallet object
val network = Network.TESTNET
val config = WalletConfig(network, "")
val wallet = Wallet.newWallet(config)

// Step 3: Create custom script using BDK scripting engine
val script = ScriptBuilder()
    .op(OP_DUP)
    .op(OP_HASH160)
    .data("abcdef1234567890abcdef1234567890abcdef12".toByteArray())
    .op(OP_EQUALVERIFY)
    .op(OP_CHECKSIG)
    .build()
```

```
23
24  // Step 4: Build and sign transaction using custom script
25  val transaction = TransactionBuilder(network)
26      .addInput(...)
27      .addOutput(...)
28      .addCustomScript(script)
29      .build()
30  val signedTransaction = wallet.sign(transaction)
31
```

## 3.5- Transaction batching

The bdk library can batch multiple Bitcoin transactions together to reduce fees and improve privacy known as transaction batching. The general steps to enable transaction batching in our wallet is:

a. Import the BDK library and configure it to work with our existing wallet codebase.
b. Create a BatchBuilder object, which will be used to create and sign the batch transaction.
c. Add inputs and outputs to the batch using the addInput() and addOutput() methods.
d. Use the buildAndSign() method to construct and sign the batch transaction.

```
import org.bitcoindevkit.bdkjni.*
import org.bitcoindevkit.bdkjni.Types.*
import org.bitcoindevkit.bdkjni.BatchBuilder.*

// Step 1: Add BDK library to project dependencies
dependencies {
    implementation("org.bitcoindevkit:bitcoindevkit:0.3.0")
}

// Step 2: Initialize BatchBuilder object
val network = Network.TESTNET
val config = WalletConfig(network, "")
val batchBuilder = BatchBuilder.newBatchBuilder(config)

// Step 3: Add inputs and outputs to batch
batchBuilder.addInput(inputTransactionId, inputTransactionOutputIndex, inputTransactionSequence)
batchBuilder.addOutput(recipientAddress, outputAmount)
batchBuilder.addOutput(changeAddress, changeAmount)

// Step 4: Construct and sign batch transaction
val signedTransaction = batchBuilder.buildAndSign(inputsToSign, keyManager)
```

## 3.6- Custom fee estimation

BDK can estimate transaction fees based on the current network conditions and suggest an appropriate fee rate for the transaction.

The general steps to enable custom fee estimation of bdk library into our wallet are:

a. Add the BDK library to our project dependencies.
b. Initialize a BDK Wallet object with the appropriate WalletConfig.
c. Use the Wallet object to estimate transaction fees.

```
import org.bitcoindevkit.bdkjni.*
import org.bitcoindevkit.bdkjni.Types.*

// Step 1: Add BDK library to project dependencies
dependencies {
    implementation("org.bitcoindevkit:bitcoindevkit:0.3.0")
}

// Step 2: Initialize BDK wallet object
val network = Network.TESTNET
val config = WalletConfig(network, "")
val wallet = Wallet.newWallet(config)

// Step 3: Use BDK wallet to estimate transaction fees
val amountToSend = 0.001 // in BTC
val recipientAddress = "bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4"
val feeRate = FeeRate.SatPerVByte(10) // custom fee rate of 10 sat/vbyte
val estimatedFee = wallet.estimateFee(amountToSend, recipientAddress, feeRate)

// Use estimatedFee in building the transaction

```

In this code, the **estimateFee** method is used to estimate the fee for a transaction with a specific amount to send, recipient address, and custom fee rate. This estimated fee can then be used when building the transaction.

## 3.7- Send to Multiple recipients

To send to multiple recipients using the BDK library, we can use the **BatchBuilder** class and its **addOutput()** method. The **addOutput()** method allows us to specify multiple outputs in a single transaction by calling the method multiple times with different output addresses and amounts.

```
import org.bitcoindevkit.bdkjni.*
import org.bitcoindevkit.bdkjni.Types.*
import org.bitcoindevkit.bdkjni.BatchBuilder.*

// Initialize BDK wallet configuration and key manager
val network = Network.TESTNET
val config = WalletConfig(network, "")
val keyManager = KeyManager(config, MemoryDatabase(), Mnemonic.encode("your mnemonic here"))

// Initialize BatchBuilder object
val batchBuilder = BatchBuilder.newBatchBuilder(config)

// Add multiple outputs to the batch transaction
batchBuilder.addOutput("output1Address", 100000L)
batchBuilder.addOutput("output2Address", 50000L)
batchBuilder.addOutput("output3Address", 25000L)

// Build and sign the transaction
val signedTransaction = batchBuilder.buildAndSign(keyManager)

```

In this code, we first initialize the BDK wallet configuration and key manager, then create a **'BatchBuilder'** object. We then add multiple outputs to the batch transaction using the **addOutput()** method with different output addresses and amounts. Finally, we build and sign the transaction using the **buildAndSign()** method.

## 3.8- Partial signatures on PSBTs

PSBT stands for Partially Signed Bitcoin Transaction, which is a format used for representing a Bitcoin transaction in a partially signed state. In a PSBT, the transaction inputs and outputs are defined, but some or all of the input signatures are missing. PSBTs are commonly used in multi-signature setups or in cases where a transaction requires multiple signatures from different parties. The PSBT format allows different parties to collaboratively sign a

transaction without sharing their private keys with each other, which increases security and privacy. PSBTs are supported by many Bitcoin wallets and software libraries, including the bdk library.

BDK's PSBT support includes the ability to create, update, and sign PSBTs, as well as extract unsigned transaction data from PSBTs. This allows for multi-party signing and advanced transaction workflows such as offline signing and hardware wallet support.

To enable PSBT (Partially Signed Bitcoin Transactions) functionality in our existing wallet using the BDK library, we can follow these general steps:

a. Integrating the BDK library into our existing **wallet** codebase.

```
// Step 1: Import the necessary BDK library classes
import org.bitcoindevkit.bdk.*
```

b. Using the BDK library's Wallet class to create a new wallet instance or retrieve an existing wallet instance.

```
// Step 2: Create or retrieve a wallet instance using the BDK library
val wallet: Wallet = Wallet.new(...)
```

c. Enabling PSBT support for the wallet by calling the **WalletConfigBuilder.psbt_support** method and passing **true** as the argument.

```
// Step 3: Enable PSBT support for the wallet
wallet.configBuilder.psbtSupport(true)
```

d. Using the **Wallet** instance to create a new PSBT transaction using the **create_tx()** method.

```
// Step 4: Create a new PSBT transaction
val txBuilder = wallet.createTx()
```

e. Sign the PSBT transaction using the **sign()** method and passing in the necessary private keys.

```
// Step 5: Sign the PSBT transaction
val signedTx = txBuilder.sign(...)
```

f. Send the signed PSBT transaction to the Bitcoin network using the **broadcast()** method.

```
// Step 6: Broadcast the signed PSBT transaction
val txid = wallet.broadcast(signedTx)
```

## 3.9- Send All functionality

The "send all" functionality in the Bitcoindevkit wallet refers to a feature that allows users to send all the available funds in their wallet to a specified recipient address, while deducting the transaction fee from the amount being sent. This can be useful for quickly and easily sending all available funds without having to manually calculate the amount to send and deduct the transaction fee separately.

To enable the "Send All" functionality in our existing wallet using the bdk library, we would need to implement the following steps:

a. Determine the total available balance of the user's wallet.
b. Subtract any transaction fees that may apply to the transaction.
c. Use the resulting value as the amount to send in the transaction.
d. Set the "send all" flag in the transaction to true.

```
// get the available balance of the user's wallet
val availableBalance = getAvailableBalance()

// subtract transaction fees
val fees = calculateTransactionFees()
val amountToSend = availableBalance - fees

// create a new transaction
val transaction = Transaction()

// set the amount to send and the "send all" flag
transaction.addOutput(amountToSend, recipientAddress)
transaction.sendAll = true

// sign and broadcast the transaction
signAndBroadcastTransaction(transaction)
```

## 3.10- Create OP_RETURN outputs

The Create OP_RETURN outputs feature in the Bitcoindevkit wallet allows the user to create a special type of transaction output called an OP_RETURN output. This output contains data that is embedded in the blockchain but cannot be spent or transferred like regular Bitcoin

outputs. OP_RETURN outputs can be used to store small amounts of metadata or to mark a transaction as belonging to a specific protocol or application.

The general steps to enable OP_RETURN output in our wallet are:

a. We first create a new TransactionBuilder object to build our transaction.
b. Then, we create an OP_RETURN output using the ScriptBuilder class and add it to our transaction using the addOutput method.

```
// Create a new transaction builder
val txBuilder = TransactionBuilder()

// Create an OP_RETURN output with a data payload
val data = "Hello, World!".toByteArray(Charsets.UTF_8)
val opReturnScript = ScriptBuilder().op(ScriptOpCodes.OP_RETURN).data(data).build()
txBuilder.addOutput(TransactionOutput(Coin.ZERO, opReturnScript))

```

## 3.11- Choose custom Electrum server

Setting up a custom Electrum server involves several steps, including configuring and running the server software, so it's not possible to provide a complete code example. However, here's a general outline of the steps involved:

1. Choose the hardware and infrastructure that we will use to run the Electrum server. This can include a server computer, hosting provider, and other resources.

2. Install and configure the ElectrumX server software on the chosen hardware or infrastructure. The ElectrumX server software is open source and can be downloaded from the official ElectrumX GitHub repository.
3. Configure the ElectrumX server to support the Bitcoin network and other requirements of our wallet app. This may involve setting up additional software or services, such as a Bitcoin node or proxy.
4. Once the ElectrumX server is configured and running, we can connect to it from our existing wallet app using the **electrum_client::Client** struct from the BDK library. Here's an example of how we can create an instance of the **Client** struct and connect to our custom Electrum server:

```
use electrum_client::{Client, ElectrumApi};
let client = Client::new("127.0.0.1:50001").unwrap();
let header = client.block_header(0).unwrap();
println!("Block header: {:?}", header);

```

In this code , we create a new **Client** instance and connect to a custom Electrum server running on **127.0.0.1:50001**. We then retrieve the **block header** for block 0 of the Bitcoin blockchain using the block_header method.

# 4- Adding features to the faucet to build its resilience so that the stress on the faucet can be reduced.

### ❖ What is a faucet

A faucet is a service provided by some cryptocurrency wallets and networks that allows users to receive a small amount of free cryptocurrency for testing or learning purposes. Faucets typically distribute small amounts of cryptocurrency at a time, often less than a dollar's worth, and require users to complete simple tasks or solve captchas to prevent bots from abusing the system. Faucets are often used by developers and enthusiasts to experiment with new cryptocurrency wallets, test transactions, or learn about the workings of a specific blockchain network.

Padawan wallet uses Tatooine Faucet so we will try to build its resilience to reduce stress.

## To reduce stress on faucet certain methods can be used:

### 4.1- Implement anti-bot measures

Bots can quickly drain the funds from a faucet, causing stress and reducing its availability for legitimate users. To prevent this, anti-bot measures can be implemented, such as CAPTCHAs, IP rate limiting, and user-agent detection.

### 4.1.1 Enabling Captcha as anti bot measure

- First, we need to sign up for the Google reCAPTCHA service and obtain a site key and a secret key. We can do this by going to the reCAPTCHA website and following the instructions.
- Once we have obtained our keys, we can add the reCAPTCHA widget to our Tatooine Faucet by adding the following HTML code to our faucet page:

```
<div class="g-recaptcha" data-sitekey="YOUR_SITE_KEY"></div>
```

- Next, we need to add the reCAPTCHA verification code to our faucet's backend. This can be done by sending a POST request to the reCAPTCHA API with the user's response and our secret key. Here's an example of how to do this in Kotlin:

```
5  val url = "https://www.google.com/recaptcha/api/siteverify"
6  val response = "USER_RESPONSE"
7  val secret = "YOUR_SECRET_KEY"
8
9  val postParams = "secret=$secret&response=$response"
10
11 val conn = URL(url).openConnection() as HttpURLConnection
12 conn.requestMethod = "POST"
13 conn.setRequestProperty("Content-Type", "application/x-www-form-urlencoded")
14 conn.doOutput = true
15 conn.outputStream.write(postParams.toByteArray(Charsets.UTF_8))
```

```
val responseCode = conn.responseCode

if (responseCode == HttpURLConnection.HTTP_OK) {
    val responseStream = conn.inputStream
    val responseString = responseStream.bufferedReader().use { it.readText() }
    val jsonResponse = JSONObject(responseString)
    val success = jsonResponse.getBoolean("success")

    if (success) {
        // Captcha verification successful, process faucet request
    } else {
        // Captcha verification failed, show error message to user
    }
} else {
    // Error occurred while verifying captcha, show error message to user
}
```

- Finally, we can add some additional measures to further protect our Tatooine Faucet from abuse. For example, we can limit the number of requests from a single IP address, implement rate limiting, or require users to complete other tasks before being able to access the faucet.

By implementing these measures, we can help ensure that our Tatooine Faucet remains resilient and is able to serve its intended purpose.

## 4.2- Implement withdrawal limits

To prevent users from taking more than their fair share of funds from the faucet, withdrawal limits can be implemented. These limits can be based on a certain time period or based on the user's IP address.

To implement a withdrawal limit on Tatooine Faucet in our wallet, we can follow these steps:

a. Determine the maximum amount of funds that can be withdrawn from the faucet by a user within a certain time period. For example, we may want to set a limit of 0.001 BTC per day.
b. Keep track of the amount of funds already withdrawn by the user. We can store this information in a database or a file.
c. When a user requests a withdrawal from the faucet, check if they have already reached the withdrawal limit. If they have, reject the withdrawal request and display an appropriate error message.
d. If the user has not reached the withdrawal limit, process the withdrawal request and update the amount of funds already withdrawn by the user.

```
val MAX_WITHDRAWAL_AMOUNT = 0.001 // Maximum withdrawal amount per day
val WITHDRAWAL_LIMIT_DURATION = 24 * 60 * 60 * 1000 // 24 hours in milliseconds

fun withdrawFromFaucet(address: String, amount: Double) {
    // Check if the user has already reached the withdrawal limit
    val withdrawalLimit = getMaxWithdrawalLimitForUser(address)
    val totalWithdrawn = getTotalWithdrawalForUser(address)
    val remainingLimit = withdrawalLimit - totalWithdrawn

    if (amount > remainingLimit) {
        throw Exception("Withdrawal amount exceeds daily limit")
    }

    // Process the withdrawal request
    sendFunds(address, amount)

    // Update the amount of funds already withdrawn by the user
    updateTotalWithdrawalForUser(address, totalWithdrawn + amount)
}

fun getMaxWithdrawalLimitForUser(address: String): Double {
    // TODO: Implement code to retrieve the maximum withdrawal limit for the user
    return MAX_WITHDRAWAL_AMOUNT
}
```

```
fun getMaxWithdrawalLimitForUser(address: String): Double {
    // TODO: Implement code to retrieve the maximum withdrawal limit for the user
    return MAX_WITHDRAWAL_AMOUNT
}

fun getTotalWithdrawalForUser(address: String): Double {
    // TODO: Implement code to retrieve the total amount of funds already withdrawn by the user
    return 0.0
}

fun updateTotalWithdrawalForUser(address: String, totalWithdrawn: Double) {
    // TODO: Implement code to update the total amount of funds already withdrawn by the user
}
```

This code snippet assumes that we have implemented the **sendFunds()** function to send funds to the user's address, and the **getMaxWithdrawalLimitForUser()**, **getTotalWithdrawalForUser()**, and **updateTotalWithdrawalForUser()** functions to retrieve and update the withdrawal limit and the amount of funds already withdrawn by the user. We can implement these functions using a database or a file to store the relevant information.

## 4.3- Implement a queue system

If the faucet has limited funds available, implementing a queue system can help reduce stress on the faucet. Users can be added to a queue and given a time estimate of when they will receive their funds.

To implement a queue system for the Tatooine Faucet in our wallet, we can follow these general steps:

a. Create a queue data structure: We can use a queue data structure, such as a FIFO (first-in, first-out) queue, to manage the requests to the Tatooine Faucet. When a user makes a request to the faucet, we add their request to the end of the queue.
b. Set a maximum number of requests in the queue: We can set a limit on the number of requests that can be in the queue at any given time. This can help prevent the queue from becoming too long and causing delays for users.
c. Process requests from the queue: we can set up a system to process requests from the queue in a timely manner.
d. Limit the number of requests per user: To prevent abuse of the faucet, we can set a limit on the number of requests each user can make within a given timeframe. This can help ensure that the faucet is available to a wide range of users.
e. Notify users of their place in the queue: To provide transparency to users, we can add a feature that notifies them of their place in the queue and estimated wait time. This can help manage expectations and prevent frustration.
f. Monitor the queue and adjust parameters as needed: It's important to monitor the queue and adjust parameters, such as the maximum number of requests and time limits, as needed to ensure that the faucet is running smoothly and not being abused.

```
import java.util.concurrent.LinkedBlockingQueue

class TatooineFaucet {
    private val queue = LinkedBlockingQueue<FaucetRequest>()
    private val faucetUrl = "https://tatooinefaucet.com/request"

    init {
        Thread {
            while (true) {
                val request = queue.take()
                // Check withdrawal limit
                if (request.amount > MAX_WITHDRAWAL_AMOUNT) {
                    request.callback.onFailure("Withdrawal amount exceeds limit")
                    continue
                }
                // Make request to faucet
                val response = makeFaucetRequest(request)
                if (response.isSuccessful) {
                    request.callback.onSuccess(response.body())
                } else {
                    request.callback.onFailure("Faucet request failed")
                }
                Thread.sleep(REQUEST_INTERVAL_MS)
            }
        }.start()
    }

    fun requestFunds(amount: Long, address: String, callback: FaucetCallback) {
        queue.put(FaucetRequest(amount, address, callback))
    }

    private fun makeFaucetRequest(request: FaucetRequest): Response {
        val requestBody = RequestBody.create(MediaType.parse("application/json"),
            "{\"address\": \"${request.address}\", \"amount\": ${request.amount}}")
        val request = Request.Builder()
            .url(faucetUrl)
            .post(requestBody)
            .build()
        return OkHttpClient().newCall(request).execute()
    }

    data class FaucetRequest(val amount: Long, val address: String, val callback: FaucetCallback)
    interface FaucetCallback {
        fun onSuccess(txid: String)
        fun onFailure(errorMsg: String)
    }

    companion object {
        private const val MAX_WITHDRAWAL_AMOUNT = 100000000L // 1 BTC
        private const val REQUEST_INTERVAL_MS = 60000L // 1 minute
    }
}
```

In this example, the TatooineFaucet class maintains a LinkedBlockingQueue of FaucetRequest objects, which contain the withdrawal amount, Bitcoin address, and a callback interface to handle the success or failure of the request. The faucet requests are processed in a separate thread that dequeues requests from the queue, checks the withdrawal limit, makes a request to the Tatooine faucet, and waits for a minute before dequeuing the next request.

To use this queue system, we would simply create an instance of the TatooineFaucet class and call its requestFunds method with the desired withdrawal amount, Bitcoin address, and a callback object to handle the response. The TatooineFaucet class takes care of adding the request to the queue and processing it in the background, so that our application can continue to function without being blocked by the faucet requests.

## 4.4- Monitor and adjust

It is important to monitor the faucet's usage and adjust its settings as necessary. For example, if the faucet is being overwhelmed by traffic, the withdrawal limits can be reduced or the anti-bot measures can be strengthened.

## ➢ Linked Technologies

1. Padawan Wallet
2. BDK library
3. Android studio
4. Kotlin
5. Java
6. Tatooine faucet

## ➢ Project Timeline

**Previous to SoB** **(April 16- April 30)**

I will continue to contribute towards the Padawan wallet and will work to resolve bugs and fix issues in the documentation.

**Program Kick-Off and Onboarding** **(May 1- May 15)**

**Week 1 (May 1 - May 7):** I plan to thoroughly understand the open-source code available for the Padawan wallet and study more about faucet and tatooine faucet.

**Week 2 (May 8- May 15):** I will try to learn more about BDK library and read its documentation to get more understanding of the project.

**First Working Period** **(May 15 - July 2)**

**Week 4 & 5 (May 15- May 29):** I plan to build the UI of app needed for future tasks and start my coding by implementing **Multi-signature support**, **Retrieving transaction history** and **Advanced fee control [Replace-by-Fee (BIP125)]** advanced feature into the wallet.

**Week 5 & 6 (May 30-June 13):** During this period I will enable **QR code scanning**, **Custom scripting**, **Transaction batching** and **Custom fee estimation** advanced features into our wallet app.

**Week 7 & 8 (June 14-July 2):** I Plan to enable **Send to multiple recipients, Partial signatures on PSBTs** and **"Send All" functionality into our wallet app.**

**Mid term Evaluations** **(July 3- July 7)**

During this period I plan to submit the work done by me to the mentor.

**Second Working Period** **(July 3- August 15)**

**Week 9 & 10(July 3- July 17):** I plan to enable **Create OP_RETURN output** and **Choose custom Electrum server** advanced features into our app wallet. Also during this period I will start working on tatooine faucet.

**Week 11 & 12(July 18- August 1):** During this period I will implement features to the faucet like anti-bot measures, withdrawal limits and queue system so that stress on the faucet will be removed.

**Closing and FInalizing** **(August 2- August 15)**

Submit complete work including documentation of Padawan wallet to mentor. And Publish the app on play store.

## ➢ Future Deliverables

1- Published app on Play store.

2- Advanced features enabled on the app.

3- Full documentation for installing and using this Project on github.

## ➢ Benefits to Community

1. Increased accessibility: The wallet makes it easier for developers to create Bitcoin applications by providing a set of pre-built components and functionalities.
2. Improved user experience: The wallet provides a user-friendly interface and supports advanced features such as coin selection, transaction batching, and custom fee estimation.
3. Open-source: The wallets are open-source, which means that anyone can inspect, audit, and contribute to the code. This fosters a collaborative and transparent development community that benefits everyone involved.

## ➢ Post SoB

I will continue contributing to Padawan wallet even after Summer of Bitcoin and will actively participate in code review and discussion in community channel. Additionally, I'm willing to work on more new projects that could benefit the community.

# ➢ Conclusion

I will always be willing to help and update the project in the near future. Also I will always be ready for debugging code since android studio is in the active development stage. So for any changes in their codebase that might reflect an error in our App, I'll be happy to modify and update them.

Also I have explained some advanced features that would not be added to padawan wallet but I would love to add them to devkit wallet anytime.

---