# Discovering new Raydium pairs

## Overview

In this article, I will present my method for detecting new X/SOL trading pairs on the Raydium (Solana blockchain) market. As I am still a beginner in the subject of Solana, this will not be an in-depth analysis, but a simple "how-to" as a good start for further research and searching for forms of optimization. Basic knowledge of Golang (code snippets will be in this language) and the Solana ecosystem is required.

The article is addressed to programmers who, like me, are taking their first steps in the Solana ecosystem, but despite basic knowledge about this ecosystem, they have trouble finding any idea to solve the following problem.

## Problem to solve

The problem we want to solve is how to automatically generate buy (potentially also sell) orders on the Raydium market for newly created pairs. To solve this problem, we need to know two things: how to detect new pairs on the Raydium market and how to automatically generate a buy order. Let's start with the latter.

## Raydium swap

Every time we buy/sell a token in the Raydium application, the appropriate transaction is generated and sent to the blockchain. An example transaction containing a buy order looks like this. The most important here is instruction #7 executed by the `Raydium Liquidity Pool V4` program (address: `675kPX9MHTjS2zt1qfr1NYHuzeLXfQM9H24wFSUt1Mp8`). Note that the statement contains multiple accounts as input arguments that we need to obtain somehow. What is known in advance is:

- Program: `675kPX9MHTjS2zt1qfr1NYHuzeLXfQM9H24wFSUt1Mp8` (Raydium program)
- InputAccount #1: `TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA` (Token program)
- InputAccount #3: `5Q544fKrFoe6tsEbD7S8EmxGTJYAKtTVhAW5Q5pge4j1` (Raydium Authority)
- InputAccount #8: `srmqPvymJeFKQ4zGQed1GFppgkRHL9kaELCbyksJtPX` (OpenBook/Serum program)

InputAccount #16, #17, #18 and arguments #19, #20 are the arguments provided by the user, and so:

- InputAccount #16: TokenAccount which SOL will be collected from (in case of purchase)
- InputAccount #17: TokenAccount to which purchased tokens will go; when we buy a token for the first time, we must first create such account; in this specific transaction, statement #6 is responsible for this
- InputAccount #18: Our wallet address
- AmountIn #19: The number of lamports we want to spend on purchasing the token; 1'000'000'000 lamports = 1 SOL
- MinAmountOut #20: What is the minimum number of tokens we expect to appear in our account; for this transaction, 0 is equivalent to setting the slippage parameter to 100%

Instruction Data is the input data of the instruction, which we decode for a given transaction as follows:

[00:01] - the first byte in the case of the `Swap` instruction always takes the value 9 (09)
[01:09] - the next eight bytes are the `AmountIn #19` argument in little-endian order (00ca9a3b00000000 is equal to 1'000'000'000)

[09:11] - the next eight bytes are the `MinAmountOut #20` argument in little-endian order (0000000000000000 is 0)

Given the above, it means that the `InputAccount #2, #4, #5, #6, #7, #9, #10, #11, #12, #13, #14, #15` arguments must be somehow obtained from the blockchain. We will do this by listening to the blockchain for relevant transactions.

Additional link to the Raydium code responsible for generating transactions in the application: https://github.com/raydium-io/raydium-sdk/blob/master/src/liquidity/liquidity.ts#L1026

It's worth taking a look after reading the entire article.

## New pool flow

Now that we know what we need to send a buy order, it's time to see an example flow of creating a new pair on Raydium.

1. Creation of a new token

This point is not important from the point of view of detecting new pairs, but additional (meta)data related to the token itself can be useful, for example, to determine whether the token was generated by a human or a script and provide information useful when deciding whether to generate a buy order. An example transaction that creates a token on the blockchain looks like this. The most important is instruction #5 of the Metaplex program, but I will not explain it in this article.

2. Creation of the Serum market (OpenBook)

This is the first transaction that must be detected to correctly identify a new pair in the Raydium market. You can check what Serum is by searching Google (for a good start: https://docs.projectserum.com/introduction/overview). In short, it is an orderbook running in the Solana ecosystem. I couldn't find any public source that provides up-to-date information about new markets, so I started analyzing the transactions responsible for creating new markets myself.

This is how example transaction responsible for creating a new market on Serum looks like. The most important instruction here is #6. This is an instruction executed by the OpenBook (Serum) program with the address `srmqPvymJeFKQ4zGQed1GFppgkRHL9kaELCbyksJtPX`. If you have already checked the input arguments for this instruction, you have probably noticed that some of the argument names overlap with those of the Raydium's `Swap` instruction. And so it is, the `InitMarket` instruction directly provides us with some of the arguments needed to execute the `Swap` instruction. These are:

- InputAccount #1: equivalent to InputAccount #9 from the `Swap` instruction (#Market -> #SerumMarket)
- InputAccount #3: equivalent to InputAccount #12 from the `Swap` instruction (#EventQueue -> #SerumEventQueue)
- InputAccount #4: equivalent to InputAccount #10 from the `Swap` instruction (#Bids -> #SerumBids)
- InputAccount #5: equivalent to InputAccount #11 from the `Swap` instruction (#Asks -> #SerumAsks)
- InputAccount #6: equivalent to InputAccount #13 from the `Swap` instruction (#BaseVault -> #SerumCoinVaultAccount)
- InputAccount #7: equivalent to InputAccount #14 from the `Swap` instruction (#QuoteVault -> #SerumPcVaultAccount)

I don't fully understand the nomenclature, but by default (based on observations of various transactions), Base/Coin corresponds to the token, and Quote/Pc corresponds to the currency (SOL, USDC, etc.). However, this is not always the case, pairs can be created the other way around and this is important for the correctness of the `Swap` instruction.

`InitMarket` also provides us with all the remaining arguments indirectly:

- VaultSignerNonce #14: used to calculate the address of InputAccount #15 from the `Swap` instruction (#SerumVaultSigner)
- InputAccount #1 (#Market): used to calculate the address of InputAccount #2, #4, #5, #6, #7 (#AmmId, #AmmOpenOrders, #AmmTargetOrders, #PoolCoinTokenAccount, #PoolPcTokenAccount)

Instruction Data is the input data of the instructions, which we decode for a given transaction as follows:

[00:05] - if I understand correctly, the first five bytes are related to the market/instruction version (via: https://github.com/project-serum/serum-dex/blob/master/dex/src/instruction.rs#L327 ) (0000000000)
[05:0D] - the next eight bytes are the `BaseLotSize #10` argument, in order little-endian (00e1f50500000000 - not important for us)
[0D:15] - the next eight bytes are the `QuoteLotSize #11` argument in little-endian order (10270000000000000 - not important for us)
[15:17] - the next two bytes are the `FeeRateBps #12` argument, in order little-endian (9600 - not important for us)
[17:1F] - the next eight bytes are the `VaultSignerNonce #14` argument in little-endian order (02000000000000000 - **needed**)
[1F:27] - the next eight bytes are the `QuoteDustThreshold #13` argument in little-endian order (f401000000000000 - not important for us) \

Having `VaultSignerNonce` we can calculate the address of `SerumVaultSigner` based on this code: https://github.com/raydium-io/raydium-sdk/blob/master/src/utils/createMarket.ts#L90

Having `SerumMarket` we can calculate the addresses of `AmmId, AmmOpenOrders, AmmTargetOrders, PoolCoinTokenAccount, PoolPcTokenAccount` using this code: https://github.com/raydium-io/raydium-sdk/blob/master/src/liquidity/liquidity.ts# L416-L559

Don't be afraid, I'll show you how to do it all in Go.

3. Adding Raydium's liquidity pool related to the Serum market

This step is not needed because we already have all the information needed to generate any valid `Swap` statement. However, add liquidity transactions are also worth listening because of additional information useful in making a decision about a buy order: the size of the pool (amount of tokens and SOL in the pool), market open time and the address to which the LP tokens will be delivered.

This is how example transaction that adds liquidity on Raydium looks like. The most important instruction here is #5. In fact, we no longer need to extract any InputAccount from this statement, and useful information is hidden in the input data to the statement.

Instruction Data is the input data to the instructions, which we decode for a given transaction as follows:

[00:01] - the first byte in the case of the `IDO Purchase` instruction always takes the value 1 (01)
[01:02] - next byte is nonce number (fe - 254)

[02:0A] - the next eight bytes are the opening time of the market; `Amount #13` as unix timestamp in little-endian order (52e995650000000000 - 1'704'323'410 - Wed Jan 03 2024 23:10:10 GMT+0000)
[0A:12] - the next eight bytes are the number of `pc` tokens in the pool in little-endian order (0060891c05600300 - 950'000'000'000'000)
[12:2A] - the next eight bytes are the number of `coin` tokens in the pool in little-endian order (00e40b5402000000 - 10'000'000'000) \

I mentioned earlier that `pc` is usually the currency (SOL) and `coin` is the token we want, but as you can see in the example of this transaction, it is exactly the opposite, so it is always worth checking the correctness of the addresses and token values in the pool.

The instruction data input layout can be found here: https://github.com/raydium-io/raydium-sdk/blob/master/src/liquidity/liquidity.ts#L1596

If we look at the code: https://github.com/raydium-io/raydium-sdk/blob/master/src/liquidity/liquidity.ts#L1598 responsible for creating the instructions, we will see that there are more accounts used to generate the transaction than shown at Solscan. This is clearly visible by opening the same transaction in Explorer: https://explorer.solana.com/tx/4CgGAnctaNtCWzsdVpVq3Vb43RnRcEGvvo7nTqA6PtHskSMpZxZxzuiTSbg6DcagB8jbLAFGdP8GUxHAhnBDf4VL

Two useful addresses are `Account #18` (address of the wallet that added liquidity) and `Account #21` (destination address for LP tokens).

## Code example

Before we start, some useful links:

- Solana RPC mainnet node configuration: https://docs.solana.com/cluster/rpc-endpoints
- RPC node http API: https://docs.solana.com/api/http
- RPC node websocket API: https://docs.solana.com/api/websocket
- Go package for Solana: https://github.com/gagliardetto/solana-go

Please note that although the code looks linear, copying each snippet one by one will not create the correct file. Each snippet is a separate logical part, showing the process of detecting new pairs step by step.

Be also aware that using single `mainnet-beta` RPC node may be insufficient because of rate limiting.

1. We will start by loading our wallet, which we will use to sign transactions:

```
package main

import (
    "os"

    bin "github.com/gagliardetto/binary"
    "github.com/gagliardetto/solana-go"
    "github.com/gagliardetto/solana-go/rpc"
    "github.com/gagliardetto/solana-go/rpc/ws"
)
```

```
func main() {
    data, err := os.ReadFile("c:/path/to/private/key/in/base.58")
    if err != nil {
        panic(err)
    }

    wallet := solana.MustPrivateKeyFromBase58(string(data))
    ...
```

2. We will then create a new websocket and rpc connection to the node to listen to the transaction logs related to OpenBook:

```
    ...
    wsClient, err := ws.Connect(ctx, rpc.MainNetBeta_WS)
    if err != nil {
        panic(err)
    }

    rpcClient, err := rpc.Connect(ctx, rpc.MainNetBeta_RPC) // Will be used later.
    if err != nil {
        panic(err)
    }

    defer wsClient.Close()

    var OpenBookDex solana.PublicKey =
solana.MustPublicKeyFromBase58("srmqPvymJeFKQ4zGQed1GFppgkRHL9kaELCbyksJtPX")
    subID, err := wsClient.LogsSubscribeMentions(OpenBookDex,
rpc.CommitmentProcessed) // CommitmentProcessed will give us information about the
transaction as soon as possible, but there is a possibility that it will not be
approved in the blockchain, so we have to check it ourselves.
    if err != nil {
        panic(err)
    }

    defer subID.Unsubscribe()
    ...
```

3. Then we will start listening (note that the Recv function is blocking, and there are so many transactions that it is worth separating the reception and processing of logs into separate goroutines):

```
    ...
    for {
        log, err := subID.Recv()
        if err != nil {
            // Handle error.
        }
```

```
        // We don't want to process logs of failed transactions.
        if log.Value.Logs == nil || log.Value.Err != nil {
            continue
        }

        foundCandidate := false

        // We parse the logs looking for potential InitMarket instructions.
        // The pattern below may not be 100% correct and may change in the future,
but for a simple example it will work.
        for i := range log.Value.Logs {
            curLog := log.Value.Logs[i]
            if !strings.Contains(curLog, "Program 11111111111111111111111111111111
success") {
                continue // Search further.
            }

            if i+1 >= len(log.Value.Logs) {
                break // No more logs.
            }

            nextLog := log.Value.Logs[i+1]
            if !strings.Contains(nextLog, "Program
srmqPvymJeFKQ4zGQed1GFppgkRHL9kaELCbyksJtPX invoke [1]") {
                continue // Search further.
            }

            // This could be InitMarket instruction.
            foundCandidate = true
            break
        }

        if !foundCandidate {
            continue
        }
        ...
```

4. Once we have found (based on the parsed logs) a possible candidate for an InitMarket transaction, we need to retrieve transaction with full details:

```
        ...
        Max_Transaction_Version := 1 // Needed to be able to retrieve non-legacy
transactions.
        ctx, cancel := context.WithTimeout(ctx, 15*time.Second)
        rpcTx, err := rpcClient.GetTransaction(rctx, log.Value.Signature,
&rpc.GetTransactionOpts{
            MaxSupportedTransactionVersion: &Max_Transaction_Version,
            Commitment:                     rpc.CommitmentConfirmed, //
GetTransaction doesn't work with CommitmentProcessed. This is also blocking
function so worth moving to separate goroutine.
        })
```

```
        cancel()

        if err != nil {
            // Handle error: retry/abort etc.
            continue
        }

        // Check if transactions failed. Don't know if its needed as we checked
that earlier on when getting logs, but doesn't cost much.
        if rpcTx.Meta.Err != nil {
            continue // Transaction failed, nothing to do with it anymore.
        }

        tx, err := rpcTx.Transaction.GetTransaction()
        if err != nil {
            continue // Error getting solana.Transaction object from rpc object.
        }
        ...
```

5. At this point, we have our transaction with all details. However, it is still a candidate. We need to analyze it to determine whether this is actually the transaction we are interested in:

```
        ...
        type market struct {
            // Serum
            Market solana.PublicKey
            EventQueue solana.PublicKey
            Bids solana.PublicKey
            Asks solana.PublicKey
            BaseVault solana.PublicKey
            QuoteVault solana.PublicKey
            BaseMint solana.PublicKey
            QuoteMint solana.PublicKey
            VaultSigner solana.PublicKey

            // Raydium
            AmmID solana.PublicKey
            AmmPoolCoinTokenAccount solana.PublicKey
            AmmPoolPcTokenAccount solana.PublicKey
            AmmPoolTokenMint solana.PublicKey
            AmmTargetOrders solana.PublicKey
            AmmOpenOrders solana.PublicKey
        }

        var m market

        for _, instr := range tx.Message.Instructions {
            program, err := tx.Message.Program(instr.ProgramIDIndex)
            if err != nil {
                continue // Program account index out of range.
            }
```

```
            if program.String() != OpenBookDex.String() {
                continue // Instruction not called by OpenBook, check next.
            }

            if len(instr.Accounts) < 10 {
                continue // Not enough accounts for InitMarket instruction.
            }

            if len(instr.Data) < 37 {
                continue // Not enough input data for InitMarket instruction.
            }

            const BaseMintIndex = 7
            const QuoteMintIndex = 8

            // Just some helper function
            safeIndex := func(idx uint16) solana.PublicKey {
                if idx >= uint16(len(tx.Message.AccountKeys)) {
                    return solana.PublicKey{}
                }
                return tx.Message.AccountKeys[idx]
            }

            // In this example we search for pairs X/SOL only.
            if safeIndex(instr.Accounts[QuoteMintIndex]) != solana.WrappedSol &&
safeIndex(instr.Accounts[BaseMintIndex]) != solana.WrappedSol {
                fmt.Printf("Found serum market, but not with SOL currency")
                break
            }

            // At this point, its probably InitMarket instruction.
            m.Market = safeIndex(instr.Accounts[0])
            m.EventQueue = safeIndex(instr.Accounts[2])
            m.Bids = safeIndex(instr.Accounts[3])
            m.Asks = safeIndex(instr.Accounts[4])
            m.BaseVault = safeIndex(instr.Accounts[5])
            m.QuoteVault = safeIndex(instr.Accounts[6])
            m.BaseMint = safeIndex(instr.Accounts[7])
            m.QuoteMint = safeIndex(instr.Accounts[8])

            caller := tx.Message.AccountKeys[0] // Additional info. Use if needed.

            // Get VaultSignerNonce from instruction data and calculate address
for SerumVaultSigner.
            vaultSignerNonce := instr.Data[23:31]
            vaultSigner, err := solana.CreateProgramAddress(
                [][]byte{
                    m.Market.Bytes()[:],
                    vaultSignerNonce,
                }, OpenBookDex)

            if err != nil {
                fmt.Printf("Error while creating vault signer account: %v", err)
```

```
                break
            }

            m.VaultSigner = vaultSigner
        }

        // Check if all accounts are filled with values. Im leaving that to the
reader.
        if !m.valid() {
            fmt.Printf("Not a InitMarket transaction")
            continue
        }
        ...
```

6. At this point we know for sure that this is an InitMarket transaction. It's time to calculate the missing addresses:

```
        ...
        // Helper function.
        associatedAccount := func(market solana.PublicKey, seed []byte)
(solana.PublicKey, error) {
            Raydium_Liquidity_Program_V4 :=
solana.MustPublicKeyFromBase58("675kPX9MHTjS2zt1qfr1NYHuzeLXfQM9H24wFSUt1Mp8")
            acc, _, err := solana.FindProgramAddress([][]byte{
                Raydium_Liquidity_Program_V4.Bytes(),
                market.Bytes(),
                seed,
            }, Raydium_Liquidity_Program_V4)
            return acc, err
        }

        ammID, err := associatedAccount(m.Market, []byte("amm_associated_seed"))
        if err != nil {
            // Handle error
        }
        m.AmmID = ammID

        ammPoolCoinTokenAccount, err := associatedAccount(m.Market,
[]byte("coin_vault_associated_seed"))
        if err != nil {
            // Handle error
        }
        m.AmmPoolCoinTokenAccount = ammPoolCoinTokenAccount

        ammPoolPcTokenAccount, err := associatedAccount(m.Market,
[]byte("pc_vault_associated_seed"))
        if err != nil {
            // Handle error
        }
        m.AmmPoolPcTokenAccount = ammPoolPcTokenAccount
```

```
        ammPoolTokenMint, err := associatedAccount(m.Market,
[]byte("lp_mint_associated_seed"))
        if err != nil {
            // Handle error
        }
        m.AmmPoolTokenMint = ammPoolTokenMint

        ammTargetOrders, err := associatedAccount(m.Market,
[]byte("target_associated_seed"))
        if err != nil {
            // Handle error
        }
        m.AmmTargetOrders = ammTargetOrders

        ammOpenOrders, err := associatedAccount(m.Market,
[]byte("open_order_associated_seed"))
        if err != nil {
            // Handle error
        }
        m.AmmOpenOrders = ammOpenOrders
        ...
```

7. At this point we have all the addresses needed to generate the swap instruction. Extracting additional information from a transaction with the `IDO Purchase` instruction is similar to the above, so I will skip it as it is not necessary. I will go straight to the stage of generating a buy order on Raydium. The code I will show here is based on the code from the repository https://github.com/gopartyparrot/goparrot-twap:

```
        ...

        // This must implement solana.Instruction interface.
        type RaySwapInstruction struct {
            bin.BaseVariant
            InAmount                uint64
            MinimumOutAmount        uint64
            solana.AccountMetaSlice `bin:"-" borsh_skip:"true"`
        }

        func NewRaydiumSwapInstruction(
            // Parameters:
            inAmount uint64,
            minimumOutAmount uint64,
            // Accounts:
            tokenProgram solana.PublicKey,
            ammId solana.PublicKey,
            ammAuthority solana.PublicKey,
            ammOpenOrders solana.PublicKey,
            ammTargetOrders solana.PublicKey,
            poolCoinTokenAccount solana.PublicKey,
            poolPcTokenAccount solana.PublicKey,
            serumProgramId solana.PublicKey,
```

```
            serumMarket solana.PublicKey,
            serumBids solana.PublicKey,
            serumAsks solana.PublicKey,
            serumEventQueue solana.PublicKey,
            serumCoinVaultAccount solana.PublicKey,
            serumPcVaultAccount solana.PublicKey,
            serumVaultSigner solana.PublicKey,
            userSourceTokenAccount solana.PublicKey,
            userDestTokenAccount solana.PublicKey,
            userOwner solana.PublicKey,
        ) *RaySwapInstruction {

            inst := RaySwapInstruction{
                InAmount:         inAmount,
                MinimumOutAmount: minimumOutAmount,
                AccountMetaSlice: make(solana.AccountMetaSlice, 18),
            }
            inst.BaseVariant = bin.BaseVariant{
                Impl: inst,
            }

            inst.AccountMetaSlice[0] = solana.Meta(tokenProgram)
            inst.AccountMetaSlice[1] = solana.Meta(ammId).WRITE()
            inst.AccountMetaSlice[2] = solana.Meta(ammAuthority)
            inst.AccountMetaSlice[3] = solana.Meta(ammOpenOrders).WRITE()
            inst.AccountMetaSlice[4] = solana.Meta(ammTargetOrders).WRITE()
            inst.AccountMetaSlice[5] = solana.Meta(poolCoinTokenAccount).WRITE()
            inst.AccountMetaSlice[6] = solana.Meta(poolPcTokenAccount).WRITE()
            inst.AccountMetaSlice[7] = solana.Meta(serumProgramId)
            inst.AccountMetaSlice[8] = solana.Meta(serumMarket).WRITE()
            inst.AccountMetaSlice[9] = solana.Meta(serumBids).WRITE()
            inst.AccountMetaSlice[10] = solana.Meta(serumAsks).WRITE()
            inst.AccountMetaSlice[11] = solana.Meta(serumEventQueue).WRITE()
            inst.AccountMetaSlice[12] = solana.Meta(serumCoinVaultAccount).WRITE()
            inst.AccountMetaSlice[13] = solana.Meta(serumPcVaultAccount).WRITE()
            inst.AccountMetaSlice[14] = solana.Meta(serumVaultSigner)
            inst.AccountMetaSlice[15] =
solana.Meta(userSourceTokenAccount).WRITE()
            inst.AccountMetaSlice[16] = solana.Meta(userDestTokenAccount).WRITE()
            inst.AccountMetaSlice[17] = solana.Meta(userOwner).SIGNER().WRITE()

            return &inst
        }

        func (inst *RaySwapInstruction) ProgramID() solana.PublicKey {
            return Raydium_Liquidity_Program_V4 // Defined in one of previous
snippets.
        }

        func (inst *RaySwapInstruction) Accounts() (out []*solana.AccountMeta) {
            return inst.Impl.(solana.AccountsGettable).GetAccounts()
        }

        func (inst *RaySwapInstruction) Data() ([]byte, error) {
```

```
            buf := new(bytes.Buffer)
            if err := bin.NewBorshEncoder(buf).Encode(inst); err != nil {
                return nil, fmt.Errorf("unable to encode instruction: %w", err)
            }
            return buf.Bytes(), nil
        }

        func (inst *RaySwapInstruction) MarshalWithEncoder(encoder *bin.Encoder)
(err error) {
            // Swap instruction is number 9
            err = encoder.WriteUint8(9)
            if err != nil {
                return err
            }
            err = encoder.WriteUint64(inst.InAmount, binary.LittleEndian)
            if err != nil {
                return err
            }
            err = encoder.WriteUint64(inst.MinimumOutAmount, binary.LittleEndian)
            if err != nil {
                return err
            }
            return nil
        }
        ...
```

8. Despite of a swap instruction, we need also to generate preceding transactions that will create a temp SOL account and a new Associated Token Account for a given token (if we are buying a token for the first time, otherwise creating an ATA is not required):

```
        ...
        tokenAddress := market.BaseMint
        instrs = []solana.Instruction{}
        signers = []solana.PrivateKey{wallet} // Wallet loaded in the first
snippet.

        // Create temporary account for the SOL that will be used as a source for
the swap.
        tempSpenderWallet := solana.NewWallet()

        // Create ATA account for the token and buyer (assume it is first buy for
the token).
        ataAccount, _, err :=
solana.FindAssociatedTokenAddress(wallet.PublicKey(), tokenAddress)
        if err != nil {
            return nil, nil, err
        }

        // Calculate how much lamports we need to create temporary account.
        // rentCost is the value returned by `getMinimumBalanceForRentExemption`
rpc call.
```

```
        // amountInLamports is user defined value (how much SOL in lamports to
spend for buying)
        accountLamports := rentCost + amountInLamports

        // Create ATA account for the token and buyer account.
        createAtaInst, err := associatedtokenaccount.NewCreateInstruction(
            wallet.PublicKey(), wallet.PublicKey(), tokenAddress,
        ).ValidateAndBuild()

        if err != nil {
            return nil, nil, err
        }

        // Create temporary account for the SOL to be swapped
        createAccountInst, err := system.NewCreateAccountInstruction(
            accountLamports,
            165, // TokenAccount size is 165 bytes always.
            solana.TokenProgramID,
            wallet.PublicKey(),
            tempSpenderWallet.PublicKey(),
        ).ValidateAndBuild()

        if err != nil {
            return nil, nil, err
        }

        // Initialize the temporary account
        initAccountInst, err := token.NewInitializeAccountInstruction(
            tempSpenderWallet.PublicKey(),
            solana.WrappedSol,
            wallet.PublicKey(),
            solana.SysVarRentPubkey,
        ).ValidateAndBuild()

        if err != nil {
            return nil, nil, err
        }

        // Create the swap instruction
        swapInst := raydium.NewRaydiumSwapInstruction(
            amountInLamports,
            minimumOutTokens,                       // This is defined by user. 0
means the swap will use 100% slippage (used often in bots).
            solana.TokenProgramID,
            market.AmmID,                           // AmmId
            Raydium_Authority_Program_V4,           // AmmAuthority
            market.AmmOpenOrders,                   // AmmOpenOrders
            market.AmmTargetOrders,                 // AmmTargetOrders
            market.AmmPoolCoinTokenAccount,         // PoolCoinTokenAccount
            market.AmmPoolPcTokenAccount,           // PoolPcTokenAccount
            serum.OpenBookDex,                      // SerumProgramId - OpenBook
            market.Market,                          // SerumMarket
            market.Bids,                            // SerumBids
            market.Asks,                            // SerumAsks
```

```
            market.EventQueue,                      // SerumEventQueue
            market.BaseVault,                       // SerumCoinVaultAccount
            market.QuoteVault,                      // SerumPcVaultAccount
            market.VaultSigner,                     // SerumVaultSigner
            tempSpenderWallet.PublicKey(),          // UserSourceTokenAccount
            ataAccount,                             // UserDestTokenAccount
            wallet.PublicKey(),                     // UserOwner (signer)
        )

        // Close temporary account
        closeAccountInst, err := token.NewCloseAccountInstruction(
            tempSpenderWallet.PublicKey(),
            wallet.PublicKey(),
            wallet.PublicKey(),
            []solana.PublicKey{},
        ).ValidateAndBuild()

        if err != nil {
            return nil, nil, err
        }

        // Set instructions and signers.
        instrs = append(instrs, createAtaInst, createAccountInst, initAccountInst,
swapInst, closeAccountInst) // Preserve order.
        signers = append(signers, tempSpenderWallet.PrivateKey)
        ...
```

9. The last step is to sign and send the transaction:

```
        ...
        // Create transaction from instructions.
        tx, err := solana.NewTransaction(
            instrs,
            latestblockhash,    // Latest Blockhash can be retrieved with
`getRecentBlockhash` rpc call.
            solana.TransactionPayer(signers[0].PublicKey()),
        )

        if err != nil {
            return nil, err
        }

        // Sign transaction.
        _, err = tx.Sign(
            func(key solana.PublicKey) *solana.PrivateKey {
                for _, payer := range signers {
                    if payer.PublicKey().Equals(key) {
                        return &payer
                    }
                }
                return nil
            },
```

```
        )

        if err != nil {
            return nil, err
        }

        MaxRetries := 5
        ctx, cancel := context.WithTimeout(ctx, 15*time.Second)
        sig, err := rpcClient.SendTransactionWithOpts(
            ctx,
            tx,
            rpc.TransactionOpts{ // Use proper options for your case.
                SkipPreflight: true,
                MaxRetries:    &MaxRetries,
            },
        )
        cancel()

        if err != nil {
            // Handle error
        }

        // Track (or not) transaction status to know it was properly stored in
blockchain.

        // Repeat process for all logs got from websocket client.
```

## Optimization possibilites

The method presented here is not optimal, but it does the job. One of the ways to optimize the entire process is to use your own or dedicated Solana node, thanks to which you will have direct access to the transaction, and the flow will change from:

Subscribe for transaction logs > receive transaction logs > parse the logs and select a candidate > send a request to RPC node for full transaction > parse transaction data

to:

Subscribe for full transactions > receive transaction > parse transaction data

## Changelog

- [17.01.2024] Initial version of document.