Solana Developer Concepts Beginner’s Guide

Solana Developer Foundations: Beginner's Concepts Guide

by ascendance.dev

by Arthur As
First Edition

To my son.
Dear reader, work hard in learning web3 and the blockchain to create the tools that your community deserves and only you can

Transaction Definition

A Solana transaction is a digital request submitted to its network for processing. It comprises instructions (program calls to change the blockchain's state), a list of involved accounts (to read or write data), and signatures (to authenticate). If an instruction fails, the entire transaction fails. Transactions enable operations like token transfers and account updates, following a specific binary format for security and consistency.

Transaction Overview

In the Solana blockchain, the execution of programs is initiated by submitting a transaction to the network. Each transaction may include one or several instructions. The Solana runtime processes these instructions sequentially and atomically within the scope of the transaction. This means that if any single instruction within a transaction encounters an error, the entire transaction is deemed unsuccessful and fails accordingly.

Accounts Definition

Solana accounts are like persistent files in a computer system, storing data beyond a single transaction's lifespan. They contain metadata dictating access permissions and are unique in having a 'lifetime' represented by lamports, a fractional native token. Accounts incur a 'rent' for occupying space in validator memory, with those dropping to zero lamports being purged. However, accounts holding enough lamports can be rent-exempt. Each account is identified by a 256-bit public key address. Accounts can be 'signers' if they authorize transactions via digital signatures. They may also be marked as 'read-only' for concurrent access, or 'executable' if they contain runnable programs. Creating an account involves registering a public key with preallocated storage space. Accounts are initially owned by the System program but can be reassigned to other programs, which then have write access. It's crucial for security that programs verify account validity to prevent malicious manipulation. Rent keeps accounts active on Solana, with a requirement for new accounts to be rent-exempt, holding enough lamports to cover two years of rent.
1 Solana is 1000000000 lamports (one billion).

Accounts Overview

Solana accounts store state between transactions, similar to files in an OS, containing data and metadata about access permissions. They differ in having a lifespan, denoted by lamports, and pay rent to remain in validator memory. An account's 256-bit public key address functions like a file path, facilitating look-up. Accounts with insufficient lamports are purged, but they can be rent-exempt if adequately funded.

Programs Definition

Solana Programs are equivalent to smart contracts in other blockchains, serving as executable code that processes transaction instructions. There are two types: Native Programs, integrated directly into the network's core, and On Chain Programs, which can be published by anyone. These programs are fundamental to Solana's operations, handling tasks like token transfers, DAO voting, and NFT ownership management. They operate on Solana's Sealevel runtime, a parallel processing model that facilitates the blockchain's high transaction speeds.

Programs Overview

Solana Programs consist of two main types: On Chain Programs and Native Programs.
On Chain Programs: User-written and akin to smart contracts in other blockchains, these programs are deployed directly onto the blockchain for public interaction and execution. They are distinct from the core Solana code (unlike Native Programs). Solana Labs maintains a subset of these in the Solana Program Library. Crucially, these programs can be updated on the blockchain by their Account owners.
Native Programs: Built into the Solana blockchain's core, Native Programs can be accessed by other programs or users. However, they can only be upgraded during core blockchain updates, with these upgrades being controlled through software releases to different clusters. Examples include the System Program (for account creation and token transfers), the BPF Loader Program (deploying, upgrading, and executing programs), and the Vote Program (managing validator voting and rewards).

Both types of programs are marked as "executable" when deployed, enabling efficient execution by Solana's runtime. Unique to Solana, these programs are upgradable: Native Programs through cluster updates and On Chain Programs by the "Upgrade Authority," typically the deployer's account.

Cross-Program Invocation Definition

Solana's Cross-Program Invocations (CPI) allow programs to call each other, enabling one program to invoke instructions of another. This process pauses the invoking program until the invoked one completes its instruction. For example, a transaction can modify accounts owned by separate on-chain programs through a CPI. Programs can also invoke instructions on behalf of a client.

CPIs are executed through the invoke() function, built into Solana's runtime. This function routes instructions to the intended program based on the program ID. However, it requires all necessary accounts for the invoked instruction, except the executable account.

Cross-Program Invocation Overview

CPIs must adhere to runtime policies, ensuring programs don't improperly modify accounts they don't own. The runtime checks account states before and after CPIs to enforce this. Additionally, CPIs can extend certain privileges, like signers and writable accounts, from the caller to the callee, based on the caller's privileges.

Programs can also use Program Derived Addresses (PDAs) to issue instructions with signed accounts not in the original transaction. PDAs are generated from seeds and a program ID, with no actual private key, making them secure for programmatic control. They allow programs to sign for addresses in CPIs, crucial for operations like transferring asset authority.

Solana restricts CPI call depth to prevent excessive nesting and limits reentrancy to direct self-recursion. These constraints maintain stability and prevent unpredictable states in programs.

In summary, Solana's Cross-Program Invocations facilitate complex interactions between programs, allowing for sophisticated operations and control flows while maintaining security and integrity within the blockchain's operational framework.

Program Derived Addresses Definition

Program Derived Addresses (PDAs) in Solana are a unique and powerful concept that enable programs to have control over specific addresses without the need for traditional private keys. Here's a more detailed explanation:

Control without Private Keys: In traditional blockchain systems, control over an address typically requires access to its private key. However, PDAs allow programs to control specific addresses without needing a private key. This is crucial for automated processes and smart contracts where human intervention (like managing private keys) is impractical or undesirable.

PDAs are the secret ingredient of Solana Programs or Smart Contracts.

How PDAs are Created

How PDAs are Created: PDAs are generated through a deterministic process using a combination of seeds (which can be any piece of data) and the program's public key. This process ensures that a PDA is unique and directly associated with a specific program. The PDA doesn't lie on the ed25519 curve used for typical Solana addresses, meaning it doesn't have a corresponding private key and cannot be controlled in the traditional sense.

Solana Validator Definition

In simpler terms, a Solana Validator is like a specialized computer that processes and validates transactions and activities on the Solana blockchain. It uses a technique called “pipelining,” which is similar to how a laundry process works in stages (washing, drying, folding) to handle multiple tasks efficiently.

Pipelining, Validator Modes, TPU, TVU

Pipelining Concept: Just like you can wash, dry, and fold different loads of laundry at the same time, pipelining in computing allows different stages of processing to happen simultaneously. This method is used to process a continuous stream of data (like transactions on a blockchain) efficiently.

Validator's Two Modes:
TPU (Transaction Processing Unit): In the TPU mode, the validator acts like a leader. It's responsible for creating new entries in the blockchain ledger, akin to writing down new transactions.
TVU (Transaction Validation Unit): In the TVU mode, the validator checks and confirms these transactions, ensuring they're valid and follow the blockchain's rules.
How it Works: Both modes use the same hardware components - like network connections, graphics processing units (GPUs), central processing units (CPUs), disk storage, and network output. However, they use these components differently depending on whether they're creating ledger entries (TPU) or validating them (TVU).
v
In summary, a Solana Validator is a crucial part of the blockchain network, ensuring that transactions are processed and validated correctly. It uses the pipelining technique to handle numerous transactions efficiently, much like handling multiple loads of laundry at different stages in a laundry cycle.

Solana Cluster Definition, Cluster Formation

A Solana Cluster is a group of validators that work together to process transactions and maintain the ledger's integrity. Here's a summary of key aspects:

Cluster Formation: To start a cluster, a genesis config is created, defining initial parameters like the number of native tokens. The first validator, known as the bootstrap validator, initializes the ledger using this config. Other validators then join by registering with the bootstrap validator or any other member of the cluster.

In summary, a Solana Cluster is a decentralized network of validators that collectively process and confirm transactions, ensuring the integrity and continuity of the Solana blockchain. The design emphasizes scalability, efficiency, and decentralization.

Pipelining, Validator Modes, TPU, TVU

Role of Validators: Validators receive data from the leader, confirm its validity through votes, and store this information. Once enough copies of the data exist across the cluster, a validator can delete its copy to save space.
Joining a Cluster: Validators join a cluster by sending registration messages through a gossip protocol. This decentralized approach ensures that all nodes eventually synchronize and have the same information, though it can be slow.
Transaction Processing: Clients send transactions to any validator. If the receiving validator isn't the leader, it forwards the transaction to the leader. The leader then timestamps and batches transactions before sending them to other validators for confirmation.
Confirming Transactions: The cluster is designed for quick confirmation times, which increase logarithmically with the number of validators. Confirmation involves the leader timestamping transactions and recognizing a supermajority of votes from validators.
Scalable Confirmation Techniques: Techniques like timestamping transactions, splitting them into batches, and using Turbine Block Propagation help manage large numbers of transactions. This approach allows scaling up to many thousands of validators.
Leader Rotation: Leaders in a Solana Cluster are rotated at fixed intervals (slots). Each leader handles transactions only in their allotted slot, ensuring an orderly process and distribution of workload.

A Solana Cluster is a group of validators that work together to process transactions and maintain the ledger's integrity. Here's a summary of key aspects:

Cluster Formation: To start a cluster, a genesis config is created, defining initial parameters like the number of native tokens. The first validator, known as the bootstrap validator, initializes the ledger using this config. Other validators then join by registering with the bootstrap validator or any other member of the cluster.

In summary, a Solana Cluster is a decentralized network of validators that collectively process and confirm transactions, ensuring the integrity and continuity of the Solana blockchain. The design emphasizes scalability, efficiency, and decentralization.

How PoH Works: Imagine a digital clock inside the blockchain. Each transaction or event gets a unique timestamp from this clock, proving exactly when it happened. This process creates a historical record that can't be altered.

The Benefit of PoH: Because each event is time-stamped, it's easier and faster to figure out the sequence of events. This reduces the need for computers in the network to talk to each other to agree on the order of transactions. It's like having a time-stamped receipt for every transaction.

Additional Features: Solana's PoH also includes algorithms for security and efficiency, like ensuring the blockchain can recover from certain types of failures and preventing fake copies of the blockchain ledger.

End Result: The combination of PoH with other technologies allows Solana to process a large number of transactions very quickly (up to 710,000 transactions per second, according to Solana whitepaper) compared to traditional blockchains.

-Node.js
-NPM
-Install ts-node: npm install -g ts-node typescript ‘@types/node’

How to run a Typescript File

On the terminal or command line run:
ts-node typescript-file.ts

@solana/web3.js is a JavaScript library providing essential tools for interacting with the Solana blockchain, enabling transaction sending, and data reading functionalities in a user-friendly manner.

Requirements:
-Node.js and NPM

Run this command on the Terminal or Command Line:

```
% npm install --save @solana/web3.js
```

JavaScript

```
JS tutorial.js > ...
1   const solanaWeb3 = require("@solana/web3.js");
2   console.log(solanaWeb3);
3
```

```
$ node tutorial.js
```

Typescript

```
TS tutorial.ts
1   import * as solanaWeb3 from "@solana/web3.js";
2   console.log(solanaWeb3);
3
```

```
$ npx ts-node tutorial.ts
```

TypeScript library @solana/spl-token

@solana/spl-token is a TypeScript library designed for interacting with the SPL (Solana Program Library) Token and Token-2022 programs on the Solana blockchain. This library is a collection of JavaScript and TypeScript bindings, which are essentially code interfaces that allow developers to interact with SPL tokens in a more straightforward manner.

Minting New SPL Tokens: The library provides functions to create (or mint) new SPL tokens. Minting is the process of generating new tokens on the blockchain.

Transferring Tokens: It offers the capability to transfer SPL tokens from one account to another. This is a fundamental feature for any token ecosystem, allowing the movement of tokens as part of transactions.

Additional Token Operations: Besides minting and transferring, the library includes functionalities for various other operations related to SPL tokens, like checking balances, managing token accounts, and so on.

In essence, @solana/spl-token simplifies the process of programming interactions with SPL tokens, making it accessible for developers to build applications that require token transactions and management on the Solana blockchain. This could include creating new tokens for a project, implementing token-based transactions in an app, or managing token assets within a digital ecosystem.

Solana Development Lifecycle

1) Prepare Your Development Environment: Begin by setting up and configuring your development environment with the necessary tools and software.

2) Craft Your Program: Engage in the creative process of writing your program, carefully coding and structuring it according to your project's requirements.

3) Compile Your Program: Convert your written program into machine-readable code by compiling it, ensuring it's ready for execution.

4) Obtain the Program's Public Address: Generate a public address for your program, which serves as a unique identifier within the network.

5) Deploy Your Program: Finally, launch your program onto the desired platform or network, making it operational and accessible for use.

Solana Development Lifecycle

To embark on Solana development with the utmost capability, it's highly recommended to install the Solana Command Line Interface (CLI) tools on your local machine. This setup provides a comprehensive and powerful development environment, equipping you with extensive tools and features for efficient Solana programming.

Alternatively, developers seeking a more streamlined approach can utilize the Solana Playground, an innovative browser-based Integrated Development Environment (IDE). It offers the convenience of writing, building, and deploying on-chain programs directly from your web browser, without the need for any software installation. This option is ideal for those who prefer an accessible, no-installation development experience.

Solana CLI

The Solana Command Line Interface (CLI) tools are a set of commands used in a terminal to interact directly with the Solana blockchain, or “cluster”. These tools are crucial for various blockchain activities, including:

Creating a Wallet: The CLI allows you to set up a new Solana wallet, which is essential for managing your digital assets on the Solana network.

Sending and Receiving SOL Tokens: With the CLI, you can execute transactions such as sending SOL, the native cryptocurrency of the Solana network, to others, or receiving it in your wallet.

Delegating Stake: The tools also enable participation in the network’s consensus mechanism by delegating your stake. This is a process where you lock up some of your SOL tokens to support the network’s operation and security, potentially earning rewards in return.

The reason for using the command-line interface is its proximity to the core development of Solana. The Solana team deploys new functionalities to the CLI first, making it the most up-to-date tool for interacting with the network. While it may not be the most user-friendly interface compared to graphical interfaces, the CLI offers the most direct, flexible, and secure way to access and manage your Solana accounts and perform a variety of blockchain-related tasks. Its use is particularly favored by developers and those who require a deeper level of control and interaction with the So-

Install Solana Tool Suite (Simplest Way)

Mac & Linux:
1)Open your Terminal App
2)Run to Install the Solana release v1.17.14:

```
sh -c “$(curl -sSfL https://release.solana.com/
v1.17.14/install)”
```

Update your PATH environment variable to incorporate the Solana programs for seamless integration and functionality

Windows:
1)Open cmd.exe as Administrator
2)Run to Install the Solana release v1.17.14:

```
cmd /c “curl https://release.solana.com/v1.17.14/
solana-install-init-x86_64-pc-windows-msvc.exe
--output C:\solana-install-tmp\solana-install-init.
exe --create-dirs”
```

Run on cmd.exe or Terminal:
solana --version

2) Craft Your Program

The development of Solana programs is predominantly conducted using the Rust programming language. These programs are crafted similarly to building a standard Rust library, utilizing Rust's robust features and syntax.

Rust Library Definition

A Rust library is a collection of code encapsulated in a crate, and can be linked to a crate, consisting of public and private functions, allowing reuse across multiple projects. It’s typically compiled with a ‘lib’ prefix in the filename.

Rust Crate Overview

Rust crates are essential modular units for Rust code, encapsulating libraries or binaries. They harness Rust's high-performance, memory-efficient capabilities and robust type system. By linking libraries into crates, developers ensure memory and thread safety, enabling powerful, secure applications with compile-time debugging, crucial for performance-critical services.

After writing the program, it needs to be compiled into Berkeley Packet Filter (BPF) bytecode, which is then ready for deployment on the blockchain After writing the program, it needs to be compiled into Berkeley Packet Filter (BPF) bytecode, which is then ready for deployment on the blockchain.

Berkeley Packet Filter

The Berkeley Packet Filter (BPF) is a technology used in various operating systems for tasks like analyzing network traffic. It provides a direct interface to data link layers, enabling the sending and receiving of raw link-layer packets. BPF allows userspace processes to define a filter program to specify desired packets, improving performance by avoiding unnecessary data transfer from the kernel. It operates through a virtual machine with instructions, often enhanced by just-in-time (JIT) compilation for efficient execution. BPF is widely used for packet filtering and other kernel functions.

With the Solana CLI, a developer creates a new Keypair for their program. The public address (or Pubkey) from this Keypair serves as the program's on-chain public address (or programId).

Using the CLI, the developer deploys the compiled program to the blockchain by sending several transactions, each carrying a small part of the program's code due to memory limits. After all parts are sent, a final transaction writes the code to the program's account, making it executable or updating it if it already exists.

Supported languages

- Rust language
- C/C++
- Python via Seahorse

Memory Safety without Garbage Collection: Rust ensures all references point to valid memory, achieving memory safety without relying on automated memory management like garbage collection.

Borrow Checker for Concurrency and Safety: Its unique ‘borrow checker’ system prevents data races by meticulously tracking object lifetimes and references during compilation, ensuring safe concurrent programming.

Functional Programming Influences: Rust incorporates elements from functional programming such as immutability, higher-order functions, and algebraic data types, enhancing its utility in systems programming and beyond.

Why Rust?

Rust's exceptional speed and memory efficiency, lacking a runtime or garbage collector, enable high-performance with critical-performance related tasks services, embedded device compatibility, and smooth integration with other languages.

Rust's advanced type system and ownership model ensure memory and thread safety, significantly reducing various types of bugs by catching them during compilation for reliable code.

Rust has a powerful community that is behind the Rustaceans thrive.

Solana Program Limitations

On-chain Rust programs can use most parts of Rust's standard libraries and many external 3rd party crates. However, they have certain restrictions due to their operation in a limited-resource, single-threaded, and using a deterministic environment.

Rust programs have no acces to these crates:
rand
std::fs
std::net
std::future
std::process
std::sync
std::task
std::thread
std::time

Limited access to these crates:
std::hash
std::os

Solana Program Limitations

Due to its high demand on computational resources and deep call stacks, Bincode should be avoided as it requires extensive processing.

Avoid string formatting as it consumes significant computational resources.

Solana Programs don’t support standard Rust print macros like println! or print!; instead, specialized Solana logging helpers must be utilized for logging purposes.

The runtime sets a cap on the number of instructions a program can execute while processing a single instruction. Refer to the computation budget for detailed information.

Solana Program Compute Budget

Each transaction on the network is assigned a compute budget to avoid overuse of resources. This budget limits the maximum compute units a transaction can use, defines costs for various operations, and sets operational boundaries to ensure efficient resource management.

During processing, a transaction uses compute units for operations like SBF instruction execution and syscalls. If it exhausts its budget or exceeds limits like call stack depth or account data size, the runtime stops the transaction and returns an error.

Operations incurring a compute cost:

-The execution of SBF instructions.
- Passing data between Solana Programs
-Calling system calls
    -logging
    -creation of program addresses
    -cross-program invocations

In Solana's cross-program invocations, the instructions called by a program inherit the resource budget from the parent program. If these instructions use up all the remaining budget of the transaction, or exceed certain limits, it leads to the termination of not only the invoked instructions but also the entire chain of invocations and the processing of the top-level transaction.

JSON RPC HTTP Methods

Solana uses the JSON-RPC 2.0 specification

Solana uses the JSON-RPC 2.0 specification for communication with nodes. This involves sending HTTP requests, typically using the @solana/web3.js library for JavaScript applications. For PubSub connections, the Websocket API is used.

Operations incurring a compute cost:

RPC HTTP Endpoint: Nodes listen on default port 8899 (e.g., http://localhost:8899).

Request Formatting: Requests are made using HTTP POST with a JSON containing four fields:

jsonrpc: “2.0”
id: A unique identifying number
method: The method name to invoke
params: An array of parameters
Response Structure: Responses are JSON objects including jsonrpc, id, and result fields, the latter holding the requested data.

Batch Requests: Multiple requests can be sent together as an array of JSON-RPC objects.

Core Definitions:
Hash: SHA-256 hash of data.
Pubkey: Public key in an Ed25519 key-pair.
Transaction: A list of instructions signed by a client keypair.
Signature: Signature of a transaction’s payload, used for identification.