Lesson 2 - Solana Theory / Rust

Lesson 1 - Blockchain / Solana architecture

Lesson 2 - Solana architecture / Rust

Lesson 3 - Rust / Solana command line tools

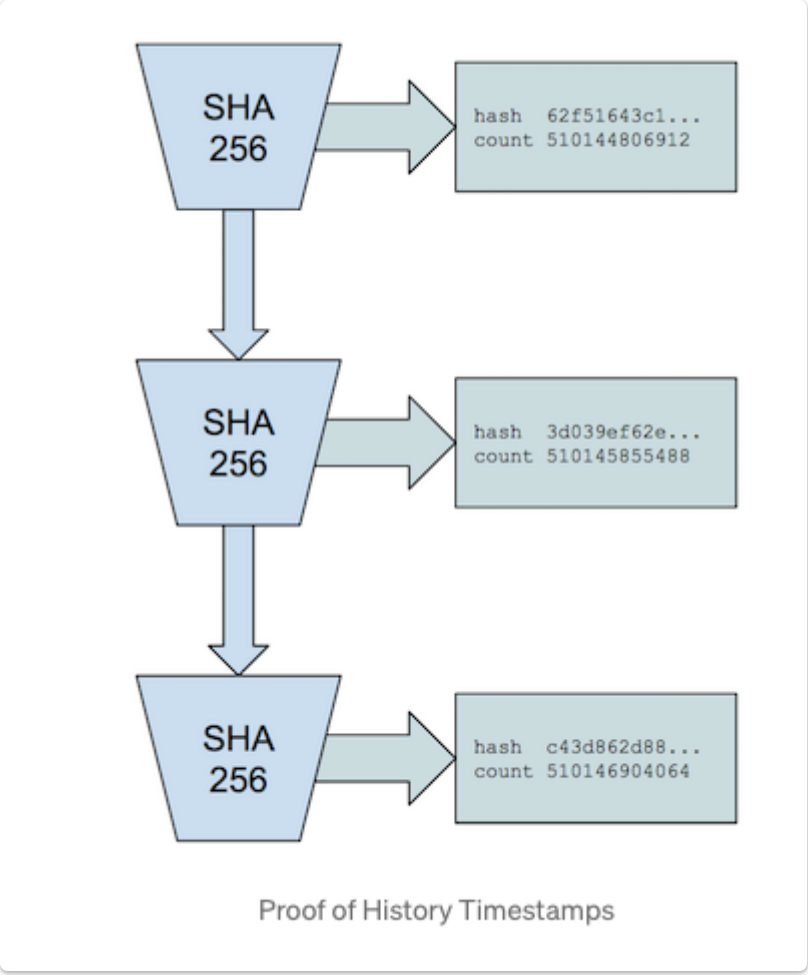
Lesson 4 - Rust / Solana development

Today's topics

- Solana Components and History
- Solana Community
- Introduction to Rust

> #### Proof of History
>
> In PoH each node can 'run its own clock' without relying on a central clock.
> From Solana docs

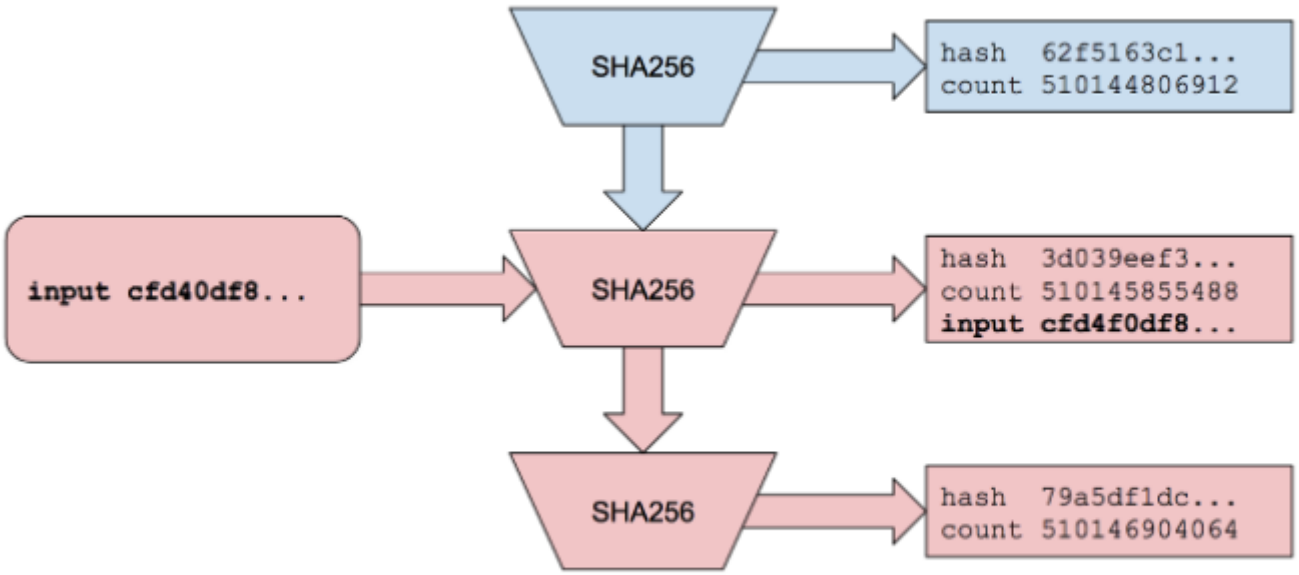
In proof of History we repeatedly hash values, the output from one step forming the input to the next step.



Proof of History Timestamps

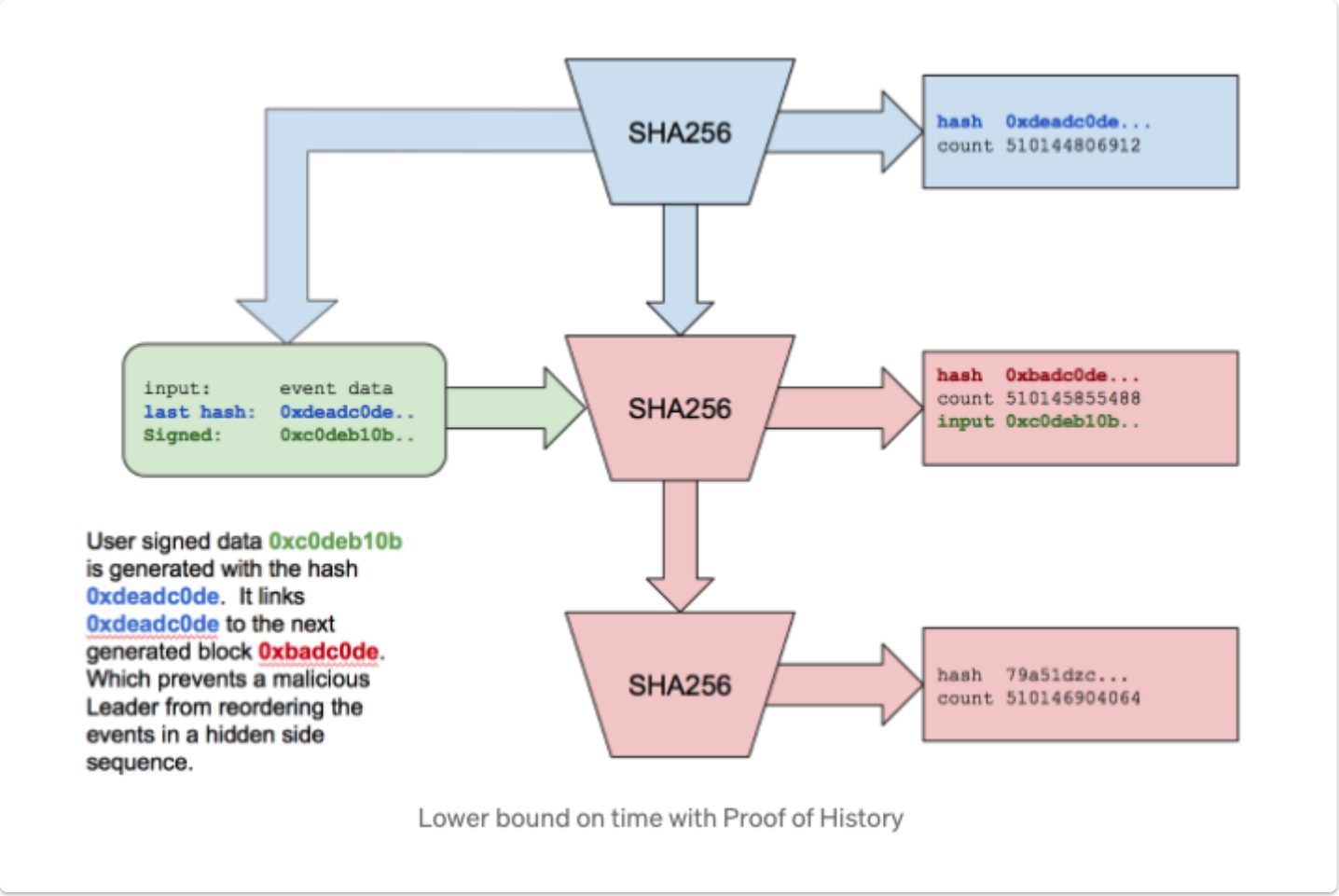
If we were given the hash values and counts out of sequence we would be able to put them into the correct order.

Upper bound on time



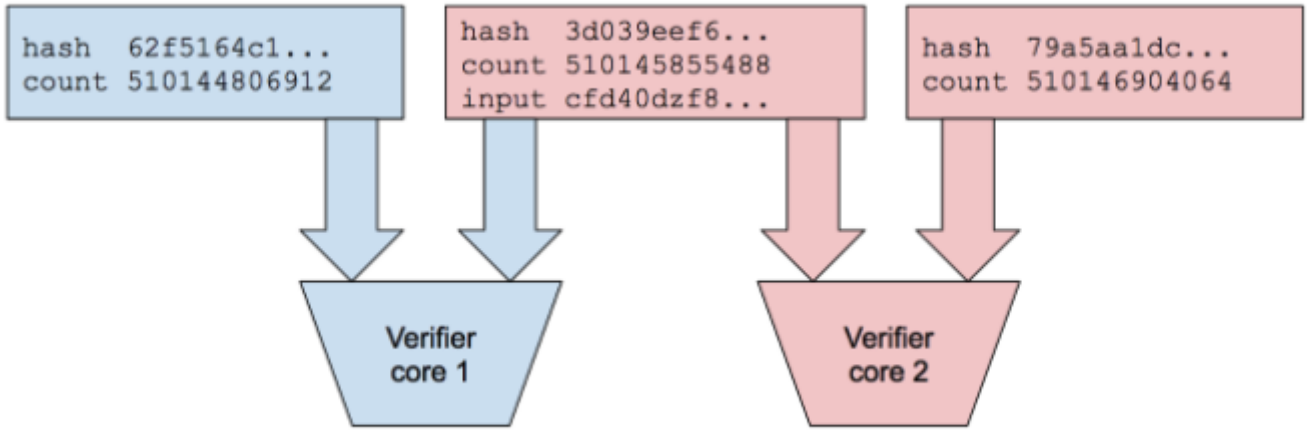
Recording messages into a Proof of History sequence

Lower Bound on time



Running this process cannot be done in parallel , we need to complete each step in turn.

However, once we have the steps we can verify in parallel.



Parallel Verification

Core 1

Index	Data	Output Hash
200	sha256(hash199)	hash200
300	sha256(hash299)	hash300

Core 2

Index	Data	Output Hash
300	sha256(hash299)	hash300
400	sha256(hash399)	hash400

A useful analogy is with a water clock, as in this article

Leader selection

Leader Schedule Generation Algorithm

Leader schedule is generated using a predefined seed. The process is as follows:

1. Periodically use the PoH tick height (a monotonically increasing counter) to seed a stable pseudo-random algorithm.
2. At that height, sample the bank for all the staked accounts with leader identities that have voted within a cluster-configured number of ticks. The sample is called the *active set*.
3. Sort the active set by stake weight.
4. Use the random seed to select nodes weighted by stake to create a stake-weighted ordering.
5. This ordering becomes valid after a cluster-configured number of ticks.

The advantages of deterministic leader selection

Since every validator knows the order of upcoming leaders, clients and validators forward transactions to the expected leader ahead of time. This allows validators to execute transactions ahead of time, reduce confirmation times, switch leaders faster, and reduce the memory pressure on validators from the unconfirmed transaction pool. This

solution is not possible in networks that have a non-deterministic leader

This system lowers latency and increases throughput because slot leaders can stream transactions to the rest of the validators in real-time rather than waiting to fill an entire block and send it at once.

As validators keep the count of time, they can stamp each incoming transaction with a time, or proof-of-history value, so the other nodes can order transactions within a block correctly even if they aren't streamed in chronological order. The other nodes can then verify these transactions as they come in rather than having to review an entire block of transactions at once.

Useful articles

Sol overview
Proof-of-history - Medium (by the founder)

Tower BFT (Proof of Stake)

Solana implements a derivation of pBFT, but with one fundamental difference. Proof of History (PoH) provides a global source of time before consensus. Solana's implementation of pBFT uses the PoH as the network clock of time, and the exponentially-increasing time-outs that replicas use in pBFT can be computed and enforced in the PoH itself.

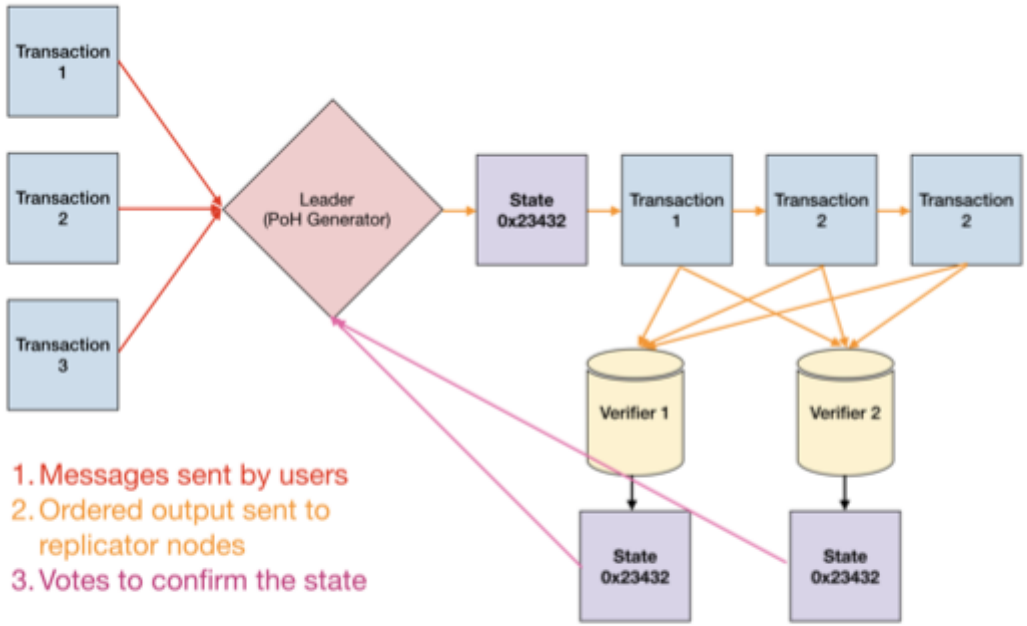


Figure 1: Transaction flow throughout the network.

The leader will be able to publish a signature of the state at a predefined period.
Each bonded validator must confirm that signature by publishing their own signed signature of the state. The vote is a simple yes vote, without a no.

If super majority of the bonded identities have voted within a timeout, then this branch would be accepted as valid.

Every 400ms, the network has a potential rollback point, but every subsequent vote doubles the amount of real time that the network would have to stall before it can unroll that vote.

Once ⅔ of validators have voted on some PoH hash, that PoH hash is canonicalized, and cannot be rolled back. This is distinct from proof of work, in which there is no notion of canonicalization.

Solana History

Anatoly Yakovenko published a whitepaper in November 2017 specifying proof of history.
The codebase was moved to Rust and became project Loom.
In Feb 2018 a throughput of > 10K transactions per second was verified.
In March 2018 the project was renamed to Solana to avoid confusion with existing projects.
In July 2018 a testnet of 50 nodes was built which managed up to 250K transactions per second.
In December 2018 the testnet was increased to 150 nodes, and the throughput averaged 200K transactions per second , peaking at 500K.
October 2020 - Wormhole bridge launched (Solana to Ethereum)
November 2022 - Solana affected by collapse of FTX, some stablecoin trading halted.
April 2023 - Solana Saga phone available

Solana Community

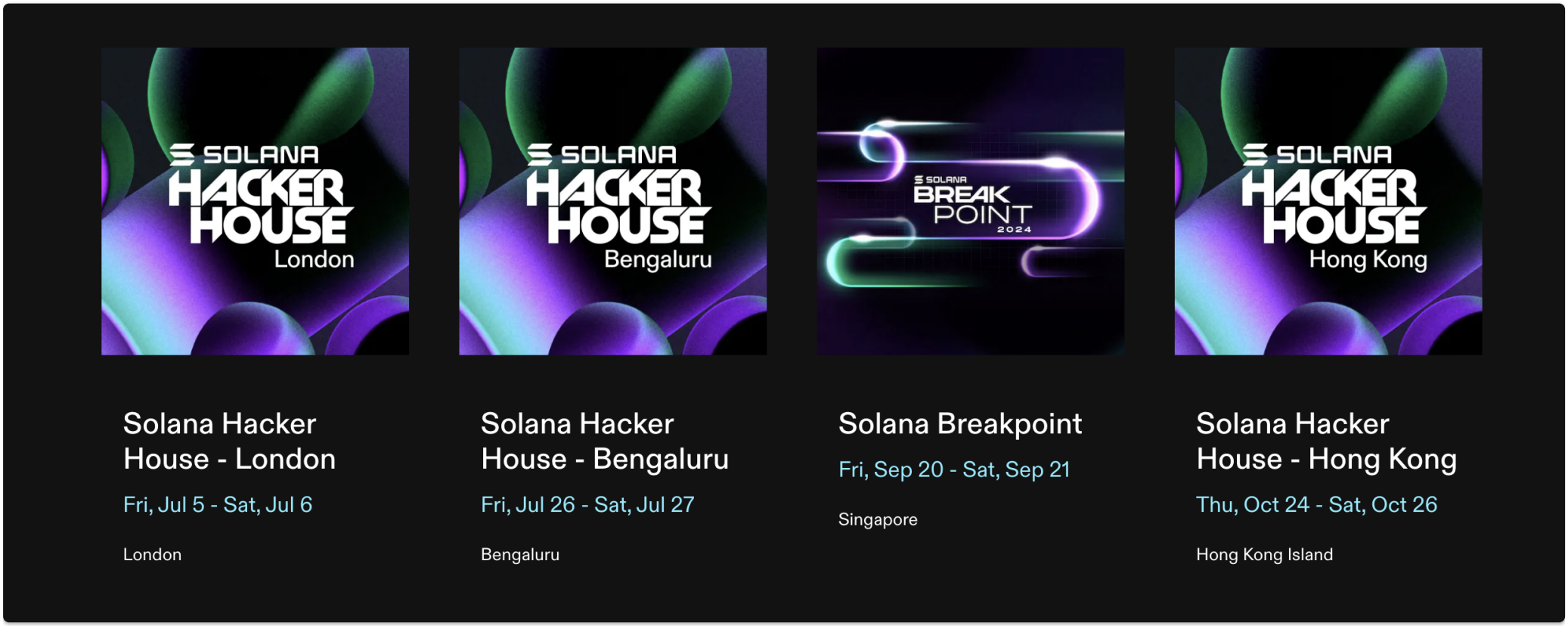
See resources page

This details their telegram / discord channels etc.

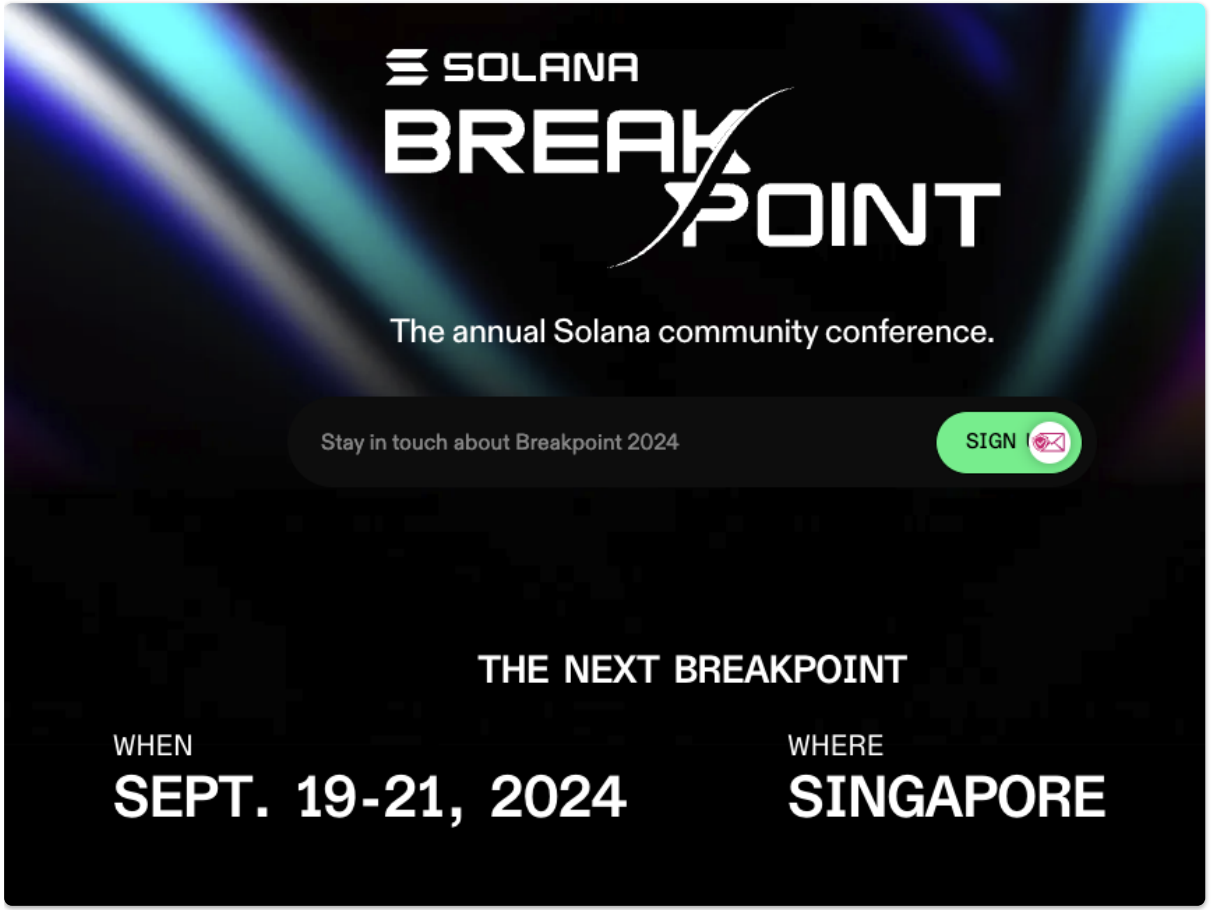
There are many meetup groups available worldwide

Hacker House

Upcoming hacker houses



Solana Breakpoint - Sept 2024



Superteam

See Site

Events

+ Submit Event

Apr 19 Friday

2:00 AM · 10:00 AM GMT+1 · 11 Sessions

London BuildStop

By Superteam UK

London, England

UK

+251

Apr 24 Wednesday

10:00 AM · 16 Sessions

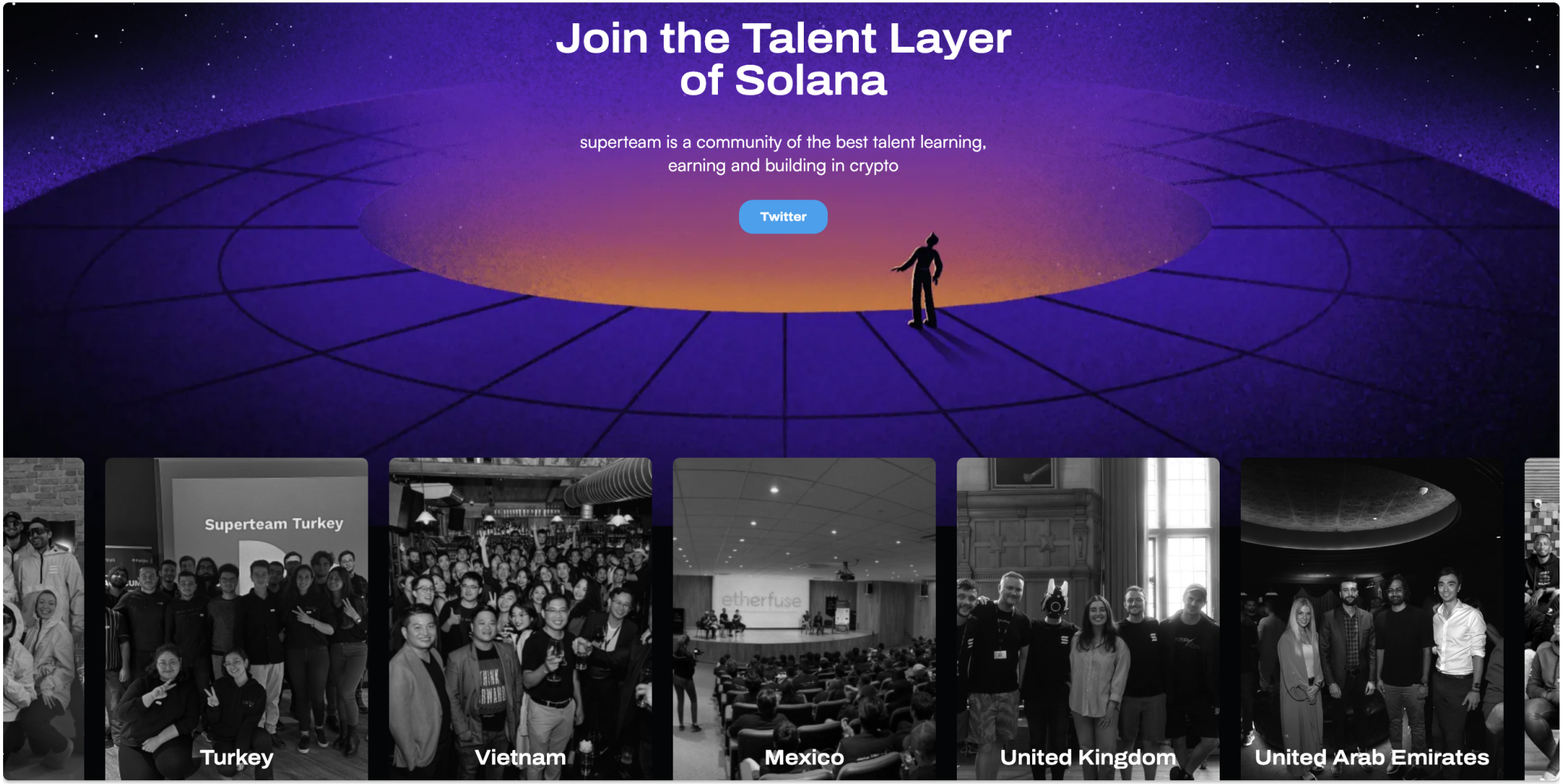
Superteam Germany Townhall

By Superteam Germany

Zoom

Germany

Global Events



Solana Collective

See Docs

This is a program to help Solana supporters contribute to the ecosystem and work with core teams.

Solana Grants

Anyone can apply for a grant from the Solana Foundation.
That includes individuals, independent teams, governments, nonprofits, companies, universities, and academics.

Here is the list of initiatives Solana are currently looking to fund. and categories they are interested in

- Censorship Resistance
- DAO Tooling
- Developer Tooling
- Education
- Payments / Solana Pay
- Financial Inclusion

- Climate Change
- Academic Research

Introduction to Rust

Core Features

- Memory safety without garbage collection
- Concurrency without data races
- Abstraction without overhead

Variables

Variable bindings are immutable by default, but this can be overridden using the `mut` modifier

```
let x = 1;
let mut y = 1;
```

Types

Data Types -Rust book

The Rust compiler can infer the types that you are using, given the information you already gave it.

Scalar Types

A *scalar* type represents a single value. Rust has four primary scalar types:

- integers
- floating-point numbers
- booleans
- characters

Integers

For example

```
u8, i32, u64
```

Floating point

Rust also has two primitive types for floating-point numbers, which are numbers with decimal points. Rust's floating-point types are `f32` and `f64`, which are 32 bits and 64 bits in size, respectively.

The default type is `f64` because on modern CPUs it's roughly the same speed as `f32` but is capable of more precision. All floating-point types are signed.

boolean

The boolean type or bool is a primitive data type that can take on one of two values, called true and false . (size of 1 byte)

char

`char` in Rust is a unique integral value representing a Unicode Scalar value
Note that unlike C, C++ this cannot be treated as a numeric type.

Other scalar types

usize

`usize` is pointer-sized, thus its actual size depends on the architecture your are compiling your program for
As an example, on a 32 bit x86 computer, `usize = u32`, while on x86_64 computers, `usize = u64`.

`usize` gives you the guarantee to be always big enough to hold any pointer or any offset in a data structure, while `u32` can be too small on some architectures.
Rust states the size of a type is not stable in cross compilations except for primitive types.

Compound Types

Compound types can group multiple values into one type.

- tuples
- arrays
- struct

Tuples

Example

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

Struct

```
struct User {
        name : String,
        age: u32,
        email: String,
}
```

Collections

- Vectors

```
let names = vec!["Bob", "Frank", "Ferris"];
```

We will cover these in more detail later

Strings

Based on UTF-8 - Unicode Transformation Format

Two string types:

- `&str` a view of a sequence of UTF8 encoded dynamic bytes, stored in binary, stack or heap. Size is unknown and it points to the first byte of the string
- `String`: growable, mutable, owned, UTF-8 encoded string. Always allocated on the heap. Includes capacity i.e. memory allocated for this string.

A String literal is a string slice stored in the application binary (i.e. there at compile time).

String vs str

`String` - heap allocated, growable UTF-8
`&str` - reference to UTF-8 string slice (could be heap, stack ...)

String vs &str - StackOverflow

Rust overview - presentation

Let's Get Rusty - Strings

Arrays

Rust book definition of an array:

"An array is a collection of objects of the same type `T`, stored in contiguous memory. Arrays are created using brackets `[]`, and their length, which is known at compile time, is part of their type signature `[T; length]`."

Array features:

- An array declaration allocates sequential memory blocks.
- Arrays are static. This means that an array once initialized cannot be resized.
- Each memory block represents an array element.
- Array elements are identified by a unique integer called the subscript/ index of the element.
- Populating the array elements is known as array initialization.
- Array element values can be updated or modified but cannot be deleted.

Array declarations

```
//Syntax1: No type definition
let variable_name = [value1,value2,value3];

let arr = [1,2,3,4,5];

//Syntax2: Data type and size specified
let variable_name:[dataType;size] = [value1,value2,value3];

let arr:[i32;5] = [1,2,3,4,5];

//Syntax3: Default valued array
let variable_name:[dataType;size] =
[default_value_for_elements,size];

let arr:[i32;3] = [0;3];

// Mutable array
let mut arr_mut:[i32;5] = [1,2,3,4,5];
```

```
// Immutable array
let arr_immut:[i32;5] = [1,2,3,4,5];
```

Rust book definition of a slice:

Slices are similar to arrays, but their length is not known at compile time. Instead, a slice is a two-word object, the first word is a pointer to the data, and the second word is the length of the slice. The word size is the same as usize, determined by the processor architecture eg 64 bits on an x86-64.

Arrays - TutorialsPoint

Arrays and Slices - RustBook

Numeric Literals

The compiler can usually infer the type of an integer literal, but you can add a suffix to specify it, e.g.
`42u8`
It usually defaults to `i32` if there is a choice of the type.

Hexadecimal, octal and binary literals are denoted by prefixes `0x` , `0o`, and `0b` respectively

To make your code more readable you can use underscores with numeric literals
e.g.

```
1_234_567_890
```

ASCII code literals

Byte literals can be used to specify ASCII codes
e.g.

`b'C'`

Conversion between types

Rust is unlike many languages in that it rarely performs implicit conversion between numeric types, if you need to do that, it has to be done explicitly.
To perform casts between types you use the `as` keyword
For example

```
let a = 12;
let b = a as usize;
```

Enums

See docs

Use the keyword `enum`

```
enum Fruit {
    Apple,
    Orange,
    Grape,
}
```

You can then reference the enum with for example

```
Fruit::Orange
```

Functions

Functions are declared with the `fn` keyword, and follow familiar syntax for the parameters and function body.

```
fn my_func(a: u32) -> bool {
    if a == 0 {
        return false;
    }
    a == 7
}
```

As you can see the final line in the function acts as a return from the function
Typically the return keyword is used where we are leaving the function before the end.

Loops

Range:

- inclusive start, exclusive end

  `for n in 1..101 {}`

- inclusive end, inclusive end

  `for n in 1..=101 {}`

    - inclusive end, inclusive end, every 2nd value

```
for n in (1..=101).step_by(2){}
```

We have already seen for loops to loop over a range, other ways to loop include

`loop` - to loop until we hit a `break`

`while` which allows an ending condition to be specified

See Rust book for examples.

Control Flow

If expressions

See Docs

The `if` keyword is followed by a condition, which *must evaluate to bool* , note that Rust does not automatically convert numerics to bool.

```
if x < 4 {
        println!("lower");
} else {
        println!("higher");
}
```

Note that 'if' is an expression rather than a statement, and as such can return a value to a 'let' statement, such as

```
fn main() {
    let condition = true;
```

```
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {}", number);
}
```

Note that the possible values of `number` here need to be of the same type.

We also have `else if` and `else` as we do in other languages.

Printing

```
println!("Hello, world!");

println!("{:?} tokens", 19);
```

Option

We may need to handle situations where a statement or function doesn't return us the value we are expecting, for this we can use Option.

Option is an enum defined in the standard library.
The `Option<T>` enum has two variants:

- `None`, to indicate failure or lack of value, and
- `Some(value)`, a tuple struct that wraps a `value` with type `T`.

It is useful in avoiding inadvertently handling null values.

Another useful enum is `Result`

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Matching

A powerful and flexible way to handle different conditions is via the `match` keyword
This is more flexible than an `if` expression in that the condition does not have to be a boolean, and pattern matching is possible.

Match Syntax

```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

Match Example

```
enum Coin {
    Penny,
    Nickel,
    Dime,
```

```
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

The keyword `match` is followed by an expression, in this case `coin`
The value of this is matched against the 'arms' in the expression.
Each `arm` is made of a pattern and some code
If the value matches the pattern, then the code is executed, each arm is an expression, so the return value of the whole match expression, is the value of the code in the arm that matched.

Matching with `Option`

```
fn main() {
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            None => None,
            Some(i) => Some(i + 1),
        }
    }

    let five = Some(5);
    let six = plus_one(five);
    let none = plus_one(None);
}
```

Installing Rust

The easiest way is via rustup

See Docs

Mac / Linux

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Windows

See details here

download and run `rustup-init.exe`.

Other methods

Cargo

See the docs

Cargo is the rust package manager, it will

- download and manage your dependencies,
- compile and build your code
- make distributable packages and upload them to public registries.

Some common cargo commands are (see all commands with --list):

```
build, b      Compile the current package
check, c      Analyse the current package and report errors,
but don't build
                          object files
clean         Remove the target directory
doc, d        Build this package's and its dependencies'
documentation
```

```
new             Create a new cargo package
init            Create a new cargo package in an existing
directory
add             Add dependencies to a manifest file
run, r          Run a binary or example of the local package
test, t         Run the tests
bench           Run the benchmarks
update          Update dependencies listed in Cargo.lock
search          Search registry for crates
publish         Package and upload this package to the registry
install         Install a Rust binary. Default location is
$HOME/.cargo/bin
uninstall       Uninstall a Rust binary
```

See `cargo help` for more information on a specific command.

Useful Resources

Rustlings

Rust by example

Rust Lang Docs

Rust Playground

Rust Book Experiment by Brown University

Rust Forum

Rust Discord