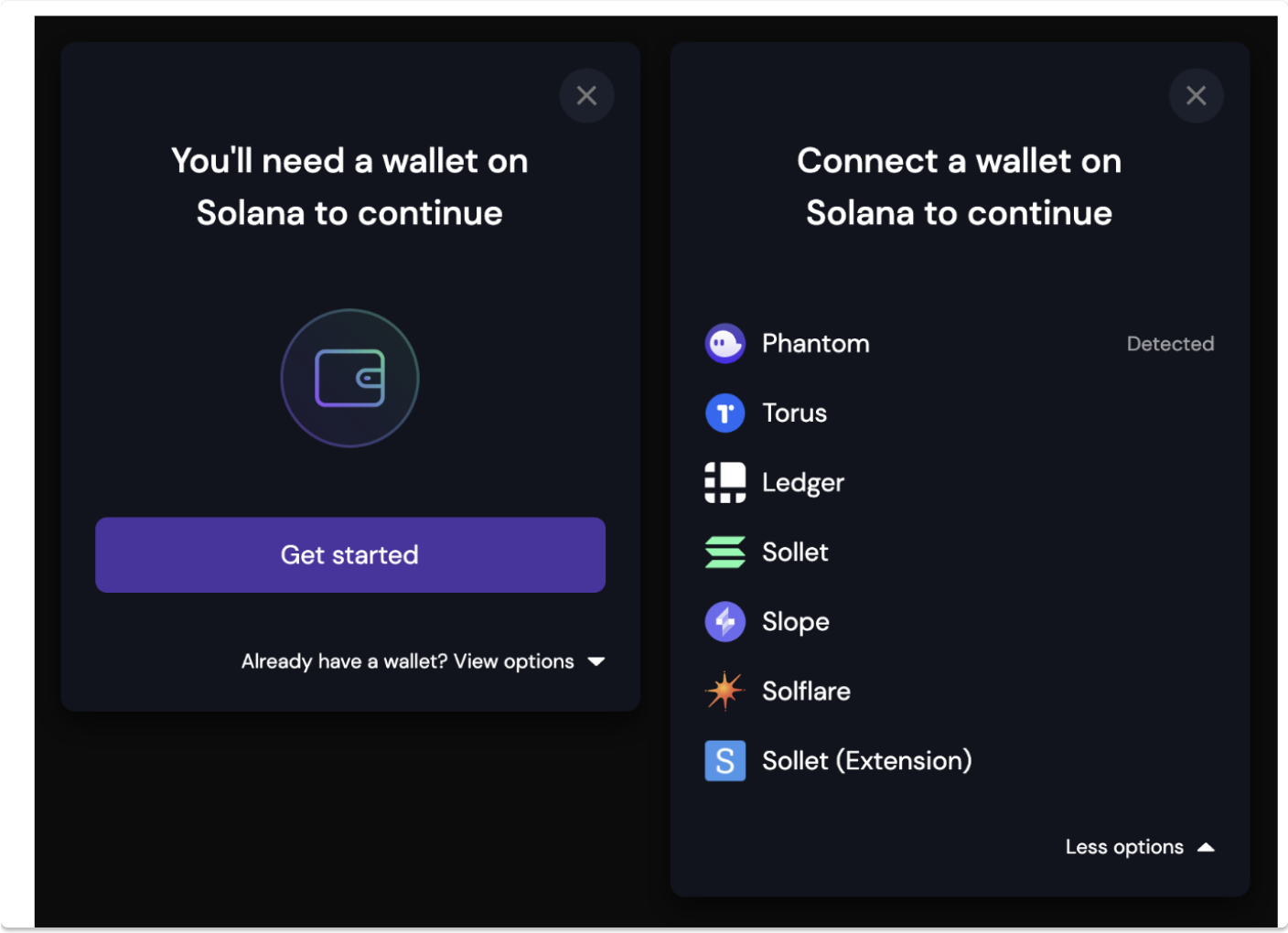
Lesson 16

Today's topics

- Wallets
- Airdrops
- Optimisation tips
- Network Congestion
- Solana Resources
- Solana Community
- Course Review
- Next Steps

Wallet Adapter Library

For external wallets you can use the [wallet adapter library]



There is a [demo] available.
There is also this [article]
For further examples see the [guide] from Magic Eden

Packages available :

You need at least the adapter base npm package

`@solana/wallet-adapter-base`

Wallet Adapter Packages

This library is organized into small packages with few dependencies.

To add it to your app, you'll need core packages, some wallets, and UI components for your chosen framework.

Core

These packages are what most projects can use to support wallets on Solana.

package	description	npm
base	Adapter interfaces, error types, and common utilities	`@solana/wallet-adapter-base`
react	Contexts and hooks for React apps	`@solana/wallet-adapter-react`

Community

Several core packages are maintained by the community to support additional frontend frameworks.

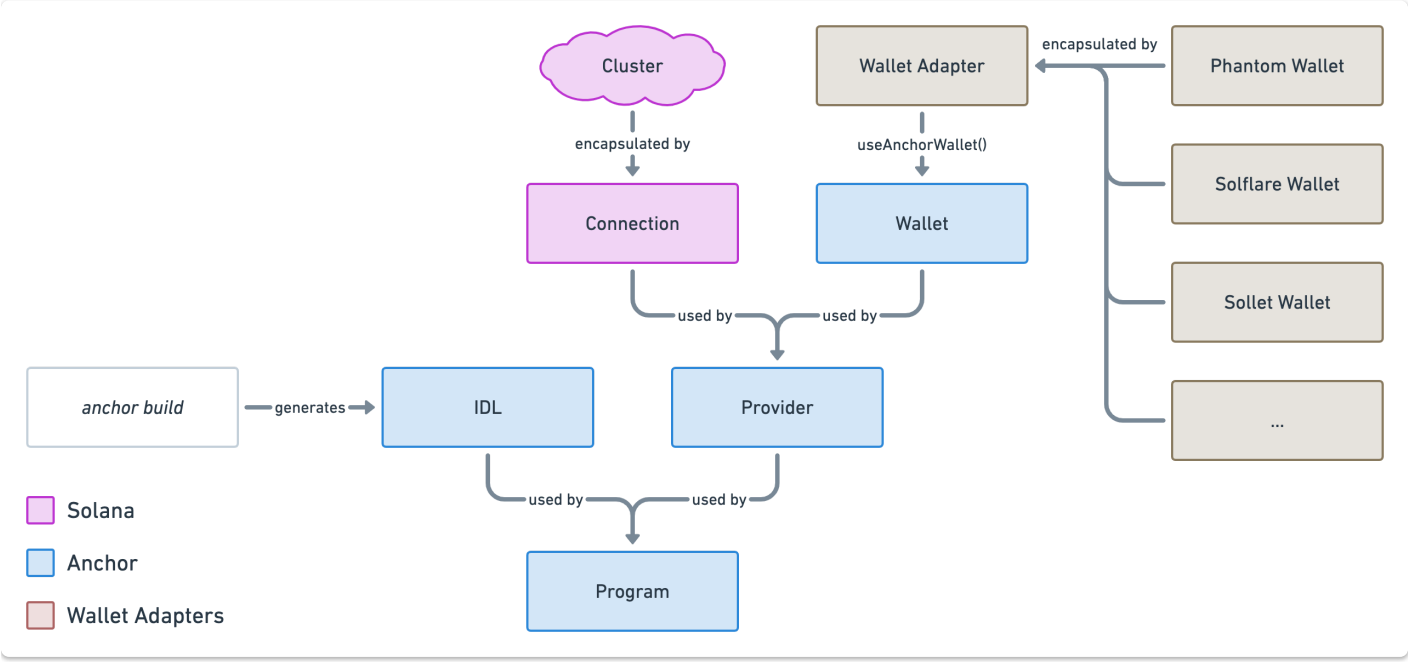
- Vue
- Angular
- Svelte
- TypeScript
- Ant Design Web3

Example code

See this quick setup guide to using the wallet adapter.

Integrating a wallet with Anchor

This article gives a good overview of how to integrate a wallet including use with Anchor. From that article



The essential difference is that you need to specify

`useAnchorWallet()`

rather than

`useWallet()`

Other options

You can use

Create Solana Dapp

For anchor support use the anchor preset

Airdrops

A naive pattern would be to transfer tokens to the entitled addresses, but if we have many addresses this could be costly. This approach is taken in this article

Another pattern we can use with airdrops, is to have a user claim the token they are entitled to. The entitlements can be stored off chain in say a database, and a UI will check their entitlement and send the token to their associated token account.
Rather than send, we may use the mint account to mint tokens to the required address.

We use Associated Token Accounts - the account we will be sending a token to.
See Airdrop tutorial

Using tools for NFTs

See this tutorial
This uses the Gumdrop tool.
Gumdrop is an NFT feature from Metaplex that allows creators to directly send users on an allowlist to a reclamation link by building the tree with off-chain handles and allowing users

to redeem into any wallet. Gumdrop can also be used with Candy Machine, complete NFT airdrops, and distribute tokens.

Gumdrop makes NFT airdrops on Solana easy by simplifying the process of dropping our Non-fungible tokens using an allowlist.

Optimisation Tips

“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming. - Donald Knuth

Be clear what circumstances you are optimising for, and decide upon an acceptable level of performance. You may have to make trade offs between different scenarios.
The correctness of the code is always the priority.
It is important to measure the compute cost, optimisation results may be counter intuitive.

See guide

- Remove unnecessary logging once it is no longer needed.
  If you need to log a public key, this is more efficient

```
// 262 cu
compute_fn! { "Log a pubkey" =>
```

```
ctx.accounts.counter.to_account_info().key
().log();
}
```

- When deriving a PDA, save the bump value
  We can use `find_program_address` to get the PDA, but this can be expensive.
  You can optimize finding the PDAs after initialization by saving the bump into an account and using it in the future.
  For example

```
pub fn pdas(ctx: Context<PdaAccounts>) ->
Result<()> {
    let program_id =
Pubkey::from_str("5w6z5PWvtkCd4PaAV7avxE6F
y5brhZsFdbRLMt8UefRQ").unwrap();

    // 12,136 CUs
    compute_fn! { "Find PDA" =>
        Pubkey::find_program_address(&
[b"counter"], ctx.program_id);
    }

    // 1,651 CUs
    compute_fn! { "Find PDA" =>
        Pubkey::create_program_address(&
```

```
[b"counter", &[248_u8]],
&program_id).unwrap();
    }

    Ok(())
}

#[derive(Accounts)]
pub struct PdaAccounts<'info> {
    #[account(mut)]
    pub counter: Account<'info,
CounterData>,
    // 12,136 CUs when not defining the
bump
    #[account(
        seeds = [b"counter"],
        bump
    )]
    pub counter_checked: Account<'info,
CounterData>,
}

#[derive(Accounts)]
pub struct PdaAccounts<'info> {
    #[account(mut)]
    pub counter: Account<'info,
CounterData>,
    // only 1600 if using the bump that is
```

```
saved in the counter_checked account
    #[account(
        seeds = [b"counter"],
        bump = counter_checked.bump
    )]
    pub counter_checked: Account<'info,
CounterData>,
}
```

- Use appropriate datatypes
  Larger datatypes are more expensive, but be aware of truncation and overflow.
- Serialisation
  See this blog about account size in Anchor
  Serialization and deserialization are both expensive operations depending on the account struct. If possible, use zero copy and directly interact with the account data to avoid these expensive operations.
  You should only use zero copy for large accounts that can not be Borsh/Anchor serialised without hitting the heap or stack limits.
  To use zero copy with an account do `#[account(zero_copy)]`

See zero copy example

Network Congestion

See article
Solana network faced congestion issues for nearly a week with a transaction failure rate as high as 75%. While developers were working on the fix, a co-founder of Solana noted that the ongoing network congestion issues were merely a bug rather than a network issue.

The Solana Foundation attributed current network congestion problems to various factors, including a large demand for Solana block space and a delayed implementation of patches to tackle network-related issues.

Improving performance

See this article

Priority Fees

See guide
These are not always implemented, but can improve the UX.
The fee is priced in micro-lamports per Compute Unit appended to transactions to induce validator nodes to include in blocks on the network.

This additional fee will be on top of the base Transaction Fee already set, which is 5000 lamports per signature in your transaction.

Guide to implementing Priority Fees

You can add to your compute units using the Compute Budget Program

```
// import { ... } from "@solana/web3.js"

const modifyComputeUnits =
ComputeBudgetProgram.setComputeUnitLimit({
  units: 300,
});

const addPriorityFee =
ComputeBudgetProgram.setComputeUnitPrice({
  microLamports: 20000,
});

const transaction = new Transaction()
  .add(modifyComputeUnits)
  .add(addPriorityFee)
  .add(
    SystemProgram.transfer({
      fromPubkey: payer.publicKey,
      toPubkey: toAccount,
      lamports: 10000000,
```

```
    }),
  );
```

Of course we need to know an appropriate amount for the priority fee, for this we can look at historic fees using the getRecentPrioritizationFees method which returns values from 150 blocks.
There are also 3rd party APIs to help, such as this from Helius

Example code from Helius

Optimize Program CU Usage: When a transaction is confirmed on the network, the transaction subtracts a number of total compute units (CU) available in a block. Today the total compute cap on a block is 48M CU, and during times of congestion this cap is often reached. Reducing the number of CUs used in your programs can increase the amount of transactions that can land on the network.
See this guide
Points to note

1. A smaller transaction is more likely to be included in a block.

2. Cheaper instructions make your program more composable.
3. Lowers overall amount of block usage, enabling more transactions to be included in a block.

Current Limits

- **Max Compute per block**: 48 million CU
- **Max Compute per account per block**: 12 million CU
- **Max Compute per transaction**: 1.4 million CU

You can measure your programs compute usage with the `compute_fn!` macro
For example

```
compute_fn!("Compute Used" => {
    // Your code here
});
```

Requesting too much compute upfront can lead to inefficient scheduling of transactions, as the scheduler does not know how much compute is left in a block until the transaction is executed. Developers should implement better scoped CU

requests that match the transaction requirements.

Solana Resources

Solana CLI Guide

See CLI Guide

The Solana Cookbook

See Cookbook

Solana Blog

See Blog

Solana Podcast

See Validated

Solana Community

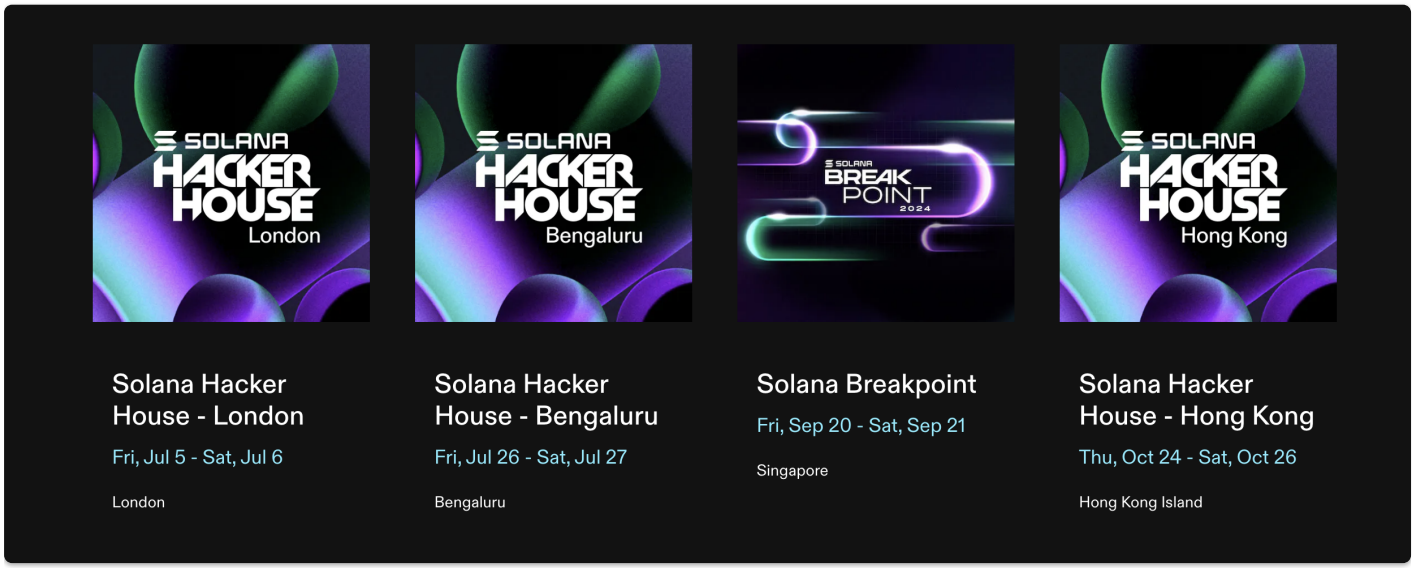
See resources page

This details their telegram / discord channels etc.

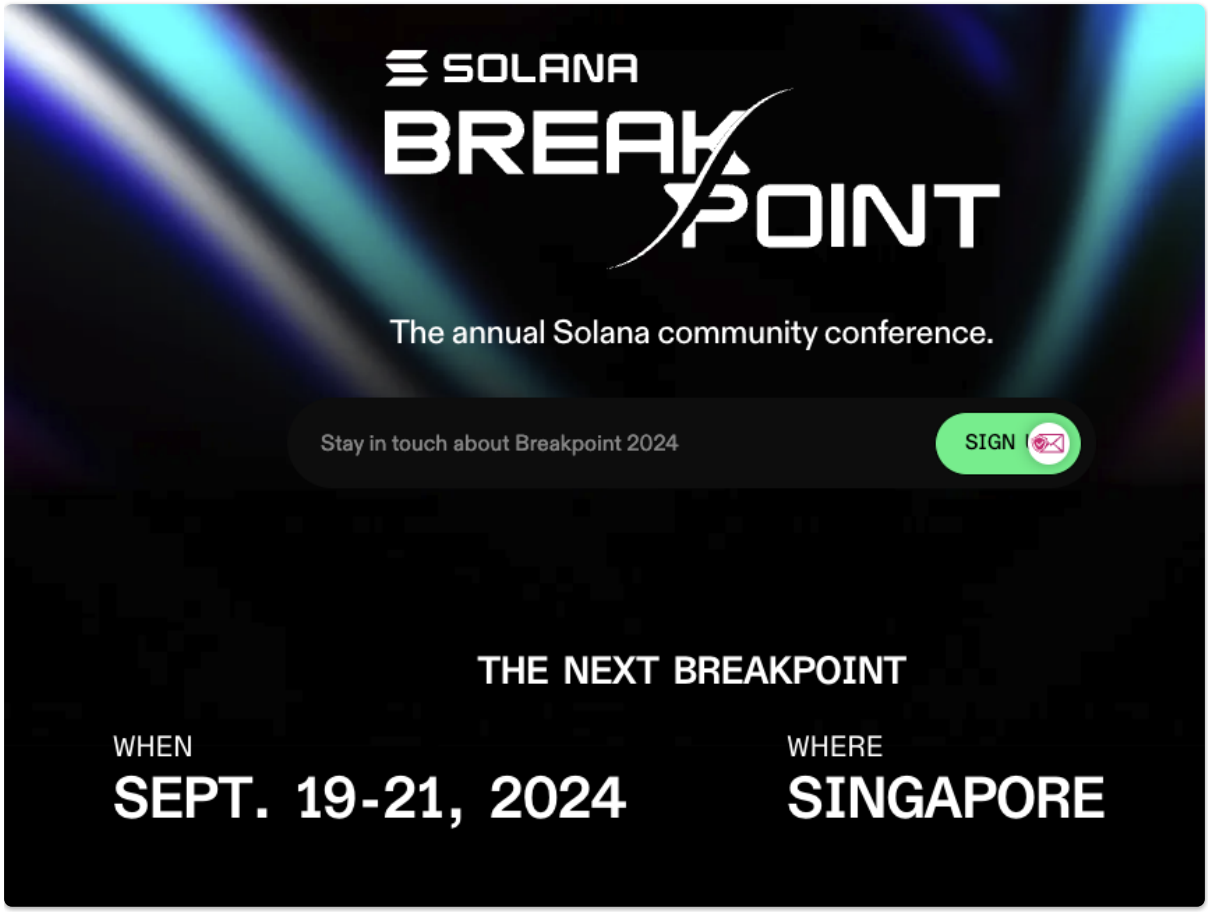
There are many meetup groups available worldwide

Hacker House

Upcoming hacker houses



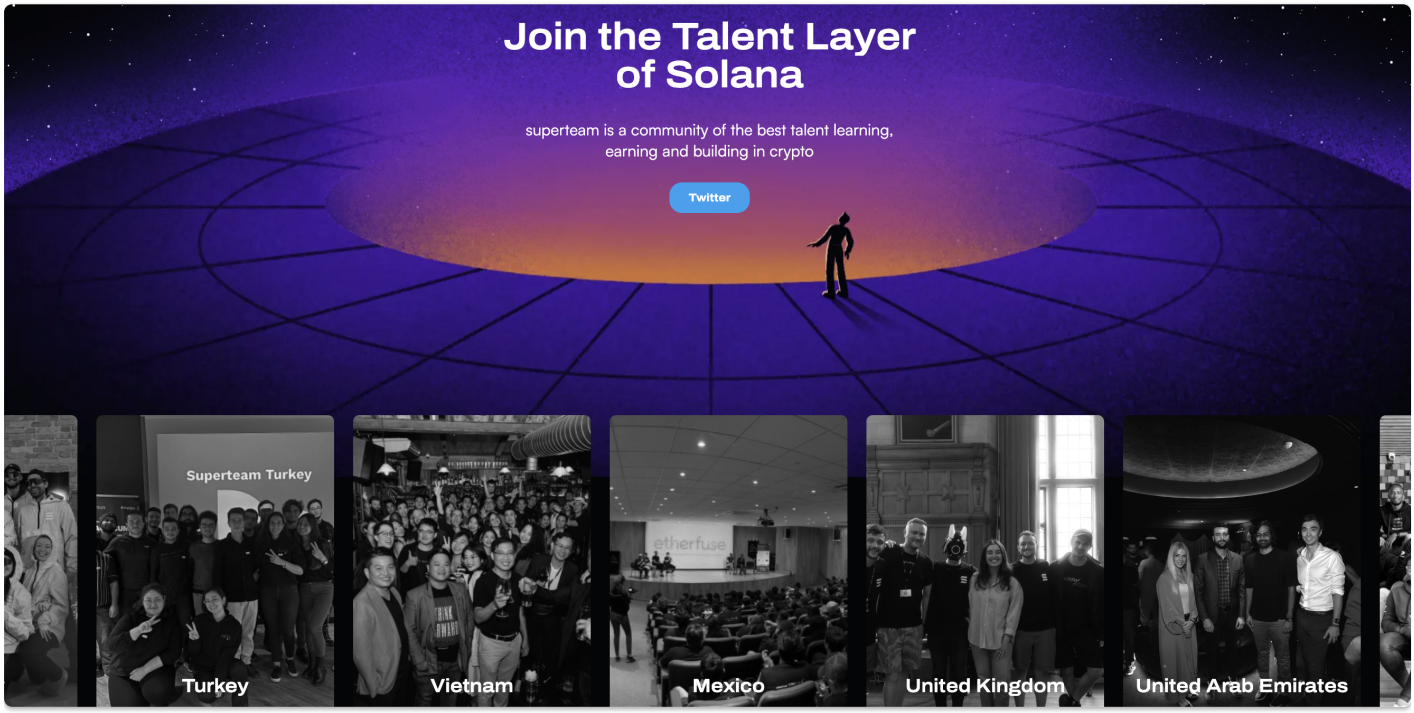
Solana Breakpoint - Sept 2024



Superteam

See Site

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

Course Review

Lesson 1

- Decentralised Systems
- Blockchain theory
- Cryptography
- Solana Architecture
- Consensus

Lesson 2

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

Lesson 3

- Solana Command Line
- Rust

Lesson 4

- Rust
- Solana Devlopment

Lesson 5

- Solana Accounts
- Programming Model
- Development Tools

Lesson 6

- Rust Lifetimes
- DeFi Introduction
- DeFi on Solana
- Tokens on Solana

Lesson 7

- Solana Programs continued
- PDAs

Lesson 8

- Authority and Ownership
- Upgrading Programs
- Program Flow
- PDAs in practice

Lesson 9

- Interface Definition Language
- Cross Program Invocation (CPI)
- Anchor Introduction

Lesson 10

Web3 Libraries

Lesson 11

- Solana Program Library
- Blockchain governance

- Solana Pay

Lesson 12

- Anchor review
- NFTs introduction
- Inscriptions
- DeFi continued

Lesson 13

- Token 2022 Program
- Interoperability / L2 solutions
- Confidential Tokens
- Security
-

Lesson 14

- Privacy on Solana
- Anchor design examples
- Versioned transactions
- Solidity and other languages

Lesson 15

- Compressed NFTs
- Debugging Solana Programs
- Pyth Oracle Network
- Eclipse

- Anchor
    - PDA and CPI
    - Further examples
- Anchor v 0.30.0
- Identity Solutions
- MEV
- Commitment Levels in Transactions

Lesson 16

- Wallets
- Airdrops
- Optimisation tips
- Network Congestion
- Solana Resources
- Solana Community
- Course Review
- Next Steps

Where to go from here

Encode Events and Bootcamps

See Events

Extropy Resources

Social Media

- X : @Extropy
- Warpcast: @Extropy
- Discord Invite
- Medium : stories