

Edgeware Network Wiki

Welcome

Announcements

Looking for Network Status or RPC Points?

→ [Networks and Public Endpoints](#)

/other-resources/networks

□ **Edgeware is Live.** You can get involved by testing functions, voting, running for elections, proposing treasury spends, upgrades, validating, nominating, or joining a Working Group - research and discussions about governance, development, branding, and more.

❗ If you haven't yet, (most have already done this, or do not need to do this) We recommend all Edgeware users convert their original Lockdrop addresses to the new network format. [See the quick steps.](#)

What is Edgeware?

Edgeware is an:

- On-chain Governed,
- Nominated Proof-of-Stake (PoS) Blockchain
- Smart Contract Platform
- with a WASM (WebAssembly) Runtime through Parity Substrate.
- And the ability to run both EVM (Solidity) Contracts, and Rust (Ink!) Contracts.

It is designed to apply decision-making process to the runtime of the blockchain, but also to the decision-making processes themselves, in an effort to rapidly generate more effective governance solutions for blockchain platforms.

Participants can vote, delegate, and fund improvements and upgrades through DAOs, economic mechanisms, cryptographic voting systems, and more.

Edgeware emphasizes user ownership and true decentralization through its governance mechanisms and through its unique token distribution mechanism, the lockdrop.

Participants in Edgeware's lockdrop staked a total of 1.1 million ETH and have a voice in the future of the network; their votes and delegations fund network upgrades.

Edgeware users value the platform for its commitment and potential to fulfill this more decentralized and user-owned vision, both for the governance mechanisms as well as its unique token distribution mechanism, the lockdrop, which distributes tokens more fairly in order to achieve more effective and representative governance.

Built on Substrate

Edgeware is built on Parity Substrate, a framework for creating new blockchains derived from the Polkadot project. Substrate implements nearly all the code necessary to launch a working blockchain, including libp2p networking, a WebAssembly runtime, PBFT consensus, and clients for running nodes and proof-of-stake validators. The engineering effort for creating Edgeware is thus limited to building the governance systems, creating well-tested compile toolchains for writing C/C++/Rust smart contracts that can be compiled to Ethereum WebAssembly (Ewasm), and porting over a pipeline for existing Ethereum Virtual Machine (EVM) smart contracts to be run on Ewasm.

Getting Involved

Edgeware is most plainly an actively-governed smart contract blockchain. Emphasis on active. Anyone can *actively*...

... participate in the lock drop. By locking ETH, you get EDG tokens which give you staking and voting rights on the new chain.

... deploy smart contracts to either a WASM or EVM virtual machine.

... participate in council and governance decisions. With stakeholders like core devs, dapp devs, end users, chain governance, and broader community stakeholders being represented by separate councils.

... develop the protocol using skills or interest that you may have. And for those efforts, be compensated with additional EDG token — representing future staking and voting rights.

This site, Edgeware Documentation, aims to become a community-developed resource for the use, validation, development on, and management of the Edgeware smart contract platform.

Why Develop on Edgeware

Edgeware is Live!

Edgeware launched in February of 2020 as the first smart-contract chain with a live mainnet. The network has transitioned through three on-chain upgrades. You can deploy your contracts to Edgeware today!

Live Full EVM + WASM Contract runtime

Edgeware has [deployed Full EVM compatibility layer](#) and tooling support. On top of that you can deploy WASM Contracts written in ink!, AssemblyScript. [Learn more how to deploy EVM and WASM contracts on Edgeware.](#)

Part of Polkadot Ecosystem

Edgeware plans to be parachain on Kusama and Polkadot, you can read [recent draft proposal for Edgeware Parachain Bonding](#). When Edgeware becomes a parachain, it will be able to leverage DOTs, KSM, ACA, Stafi Derivatives within Edgeware DeFi projects, integrate with privacy layers and other oracle solutions.

Low Transactions Fees

Due different and better architecture than Ethereum, [Edgeware has low transaction fees](#).

Ethereum EVM bridge

On-going efforts to migrate dApps from Ethereum network to Edgeware and have same dApp experience like on Ethereum. [You can deploy your dApp on Edgeware today.](#)

Large and Active Community

Over 1.2m ETH participated in our lockdrop, with funds, individuals, and projects participating. You'll be releasing your project to an active ecosystem. 8k active addresses with ~10 million [to receive EDG through partners](#). Edgeware has nowadays 100 daily active participants across voting, discussion and development and it's growing.

Funding & Incentives

- \$5M plus in funding available per year in grants, investments and more
 - Generally [Edgeware has construction-projects](#) for Treasury funding available for your idea, tooling or existing project. You can request amount up to \$100k per project.
 - BuildDAO - [aims to be an on-chain incubator](#). Apply with an idea, get \$50k to seed your startup or protocol on Edgeware
 - [Experimental Developer Mining](#) - Earn usage based rewards of fees generated by your smart contract. Potentially 2% of fees generated by your smart contract.
 - [Developer in Residence program](#) - for individuals who want to leverage their skills and have ownership over their time and work. That's include producing tooling, product or protocol
-

Growing Ecosystem

Several venture-backed projects newly developed on Edgeware, a growing ecosystem of community participants and more.

Working Groups

At Commonwealth, you can find vibrant working groups that operate on top of Edgeware. You can find here categories like DeFi, Gaming, Social/DAOs and Tooling & Ecosystem. You can engage with focused community members through chats

Advanced Developer Features

- Zero-knowledge primitives
- Anonymous group actions

Roadmap

2020

Increasing Validator Count via Governance Action

- Currently 60 to 85 proposed.
- Promotes network security and decentralization.

Adding Registrars to Improve On-Chain ID verification of Important Figures, including Validators and Council

- First Registrar Added March 17

Smart Contract Functionality

- Dependent upon Development of Compilers and the Ink! Smart Contract Language
- Native Ink! Utilizes Smart Contract Substrate Palette
- Palette may also support Solidity Wasm.

Onboard the Ethereum Ecosystem, Dapps and Developers

- Support for Ethereum Dapps and Solidity
- Ethereum <> Edgeware/Polkadot Bridge (depending on Parachain status)

VM / Platform	Methods
Ethereum EVM	Substrate EVM (Solidity/EVM,) Solang (Solidity to Substrate Compiler),
Ethereum Bridge	Centrifuge Bridge, ChainX, ChainSafe ChainBridge
Ethereum EWasm	Second State Ewasm VM

Assess Opportunity on Polkadot

- Polkadot expected to launch in 2020.
- Will require that the community approve and plan to acquire DOTs for parachain slot auction.

Track and Assess Substrate Improvements for Upgrade

2021

Quickstart

Retrieve your Locked ETH

There are two ways to retrieve your ETH from a lockdrop user contract:

- Using the Commonwealth unlock tool. (This tool is down occasionally.)
- Sending a manual transaction.

First, note the difference between a LUC and the MLC:

- **Master Lockdrop Contract (MLC) (v1 and v2)**

The coordinating contract that you sent your ETH to - it creates and sends this ETH on to your personal Lockdrop User Contract, which holds your ETH until unlock time. **Do not send the unlock transaction to an MLC - it will fail.**

- MLC v1 (Old) `0x1b75b90e60070d37cfa9d87affd124bb345bf70a`
- MLC v2 (New) `0xFEC6F679e32D45E22736aD09dFdF6E3368704e31`
- **Lockdrop User Contract (LUC)** Your individual lockdrop contract. This holds the ETH and is the contract you will use to unlock and retrieve your ETH. **This is where we will send an unlock transaction. See link below to find your LUC address.**

Has your lock duration elapsed?

To retrieve your ETH, your lock duration must be complete or the transaction will fail. Before proceeding with this guide, check that your duration is over:

→ **Check the Status of Your Lock Duration and Unlock Date**

</edge-ware-runtime/lockdrop/check-the-status-of-your-lock-duration-and-unlock-date>

Prerequisites

- Access and control over the *original* account you sent from when creating your LUC from the MLC - the ETH will be returned here.
- **Metamask** or a wallet with transaction abilities with *any* account connected, it *does not* have to be the original, but it **does** need enough ETH to pay the 'gas' (the transaction fee) for the transaction - likely equal or less than 0.00042 ETH.
- The *contract* address of your LUC. See above to learn how to find this.
- The unlock time must have passed for your LUC.

Steps

1. Open Metamask or wallet and select the account you want to pay the transaction fee from.
2. Hit the Send Button
3. Confirm that the account Selected at top has enough ETH to pay the transaction fee.
4. Carefully enter your LUC's Contract Address into the Recipient Address field.
5. Enter zero for the send amount.
6. Increase your gas limit on a transaction to 40k.
7. Confirm that your transaction speed (and therefore transaction fee cost) is satisfactory. (More fee means quicker processing by the network.)
8. Hit Next



Unlock attempts fail due to three main reasons:

- The LUC is not yet past its unlock date of 3, 6 or 12 months.
- The gas limit needs to be increased to 40k.
- You are attempting to send the transaction to an MLC and not your LUC.

Now watch for the transaction to be finalized, and confirm that your ETH arrived home safely.

Prerequisites

- The ETH address of the account you participated in the lockdrop with.
- Control over the account you originally participated in the lockdrop with - the ETH will return here.
- A web3 Wallet with an account with enough ETH to cover the unlock transaction (gas) costs.
- Your unlock time has elapsed.

 This tool may be offline or display errors due to Infura - if so, switch to the manual process in this guide.

Steps

Visit the [Unlock tool at Commonwealth.im](#)

Select the version of the Master Lockdrop Contract you used to participate in the Lockdrop event.

- MLC v1 (Old) 0x1b75b90e60070d37cfa9d87affd124bb345bf70a
- MLC v2 (New) 0xFEC6F679e32D45E22736aD09dFdF6E3368704e31

If you are not sure which you used, examine the transactions at your participating ETH address using a block explorer for the locking transactions.

Enter your participating ETH address and hit "Get Locks" button.

View the LUC instances and select unlock to cue a Metamask or web3 wallet transaction. Send the transaction.

 Unlock attempts fail due to two main reasons:

- The LUC is not yet past it's unlock date of 3, 6 or 12 months.
- The gas limit needs to be increased to 40k.

Create an Account

An address is the public part of a Edgeware account. The private part is the key used to access this address. The public and private part together make up a Edgeware account. To interact with Edgeware chain create such as creating basic transactions and various operation, you need to have created Account.

There are several ways to generate a Edgeware account:

- [Polkadot{js} Browser Plugin](#) - *We recommend this for most users*
 - [Subkey](#) - *Advanced and Most secure*
 - [DotApps.io](#) (known as Polkadot-JS Apps)
 - Parity Signer
 - Ledger Hardware Wallet
-

Storing your key safely

The seed is your **key** to the account. Knowing the seed allows you, or anyone else who knows the seed, to re-generate and control this account.

It is imperative to store the seed somewhere safe, secret, and secure. If you lose access to your account (i.e. you forget the password for your account's JSON file), you can re-create it by entering the seed. This also means that somebody else can have control over your account if they have access to your seed.

For maximum security, the seed should be written down on paper or another non-digital device and stored in a safe place. You may also want to protect your seed from physical damage, as well (e.g. by storing in a sealed plastic bag to prevent water damage, storing it in a fireproof safe, etching it in metal, etc.) It is recommended that you store multiple copies of the seed in geographically separate locations (e.g., one in your home safe and one in a safety deposit box at your bank).

You should definitely not store your seed on any kind of computer that has or may have access to the internet in the future.

Storing your account's JSON file

The JSON file is encrypted with a password, which means you can import it into any wallet which supports JSON imports, but to then use it, you need the password. You don't have to be as careful with your JSON file's storage as you would with your seed (i.e. it can be on a USB drive near you), but remember that in this case your account is only as secure as the password you used to encrypt it. Do not use easy to guess or hard to remember passwords. It is good practice to use a [mnemonic password of four to five words](#). These are nearly impossible for computers to guess due to the number of combinations possible, but much easier for humans to remember.

Polkadot{.js} Browser Plugin

The Polkadot{.js} plugin provides a reasonable balance of security and usability. It provides a separate local mechanism to generate your address and interact with Polkadot.

This method involves installing the Polkadot{.js} plugin and using it as a "virtual vault," separate from your browser, to store your private keys. It also allows signing of transactions and similar functionality.

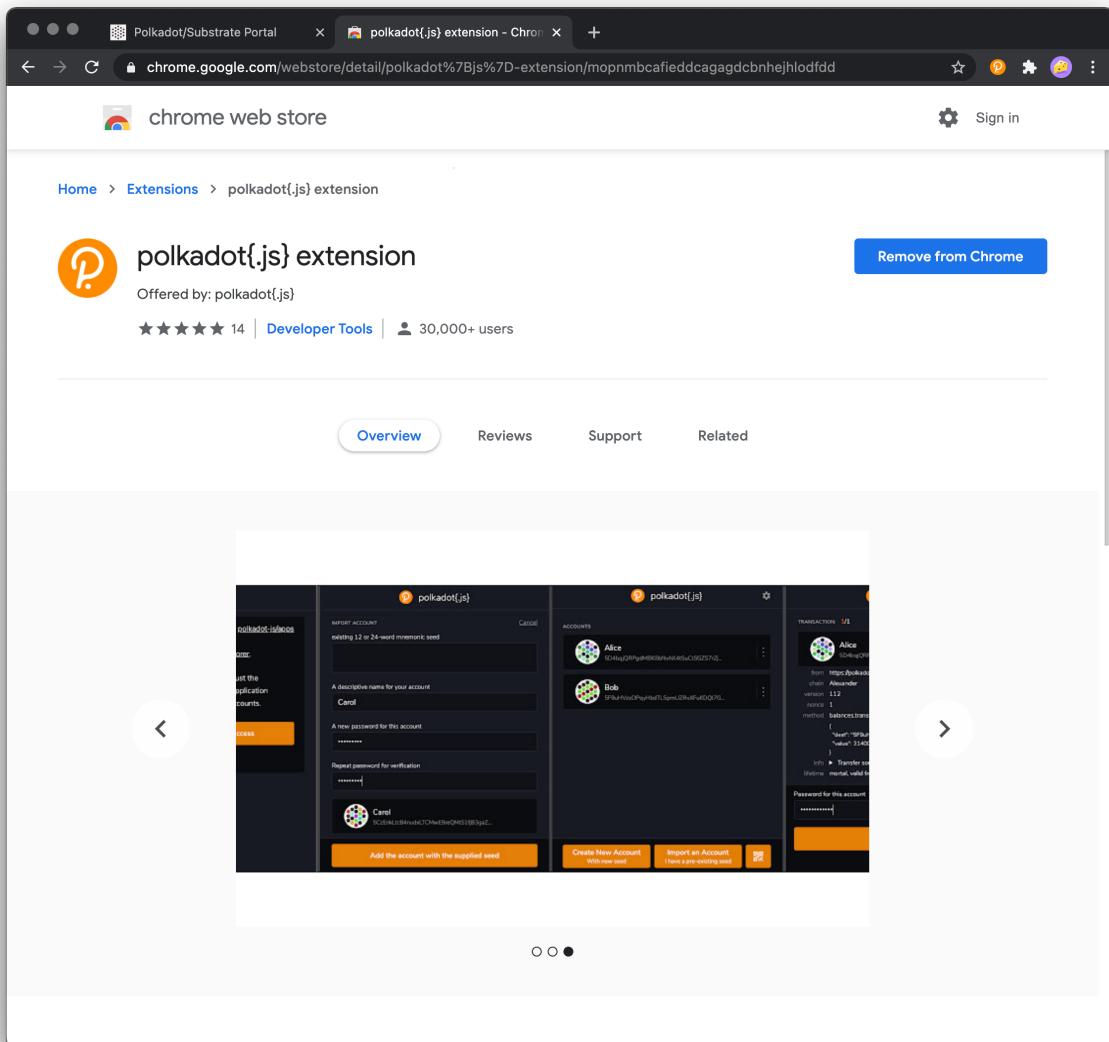
It is still running on the same computer you use to connect to the internet with and thus is less secure than using Parity Signer or other air-gapped approaches.

Install the Browser Plugin

The browser plugin is available for [both Google Chrome \(and Chromium based browsers like Brave\)](#) and [FireFox](#).

If you would like to know more or review the code of the plugin yourself, [you can visit the Github source repository](#).

After installing the plugin, you should see the orange and white Polkadot{.js} logo in the menu bar of your browser.



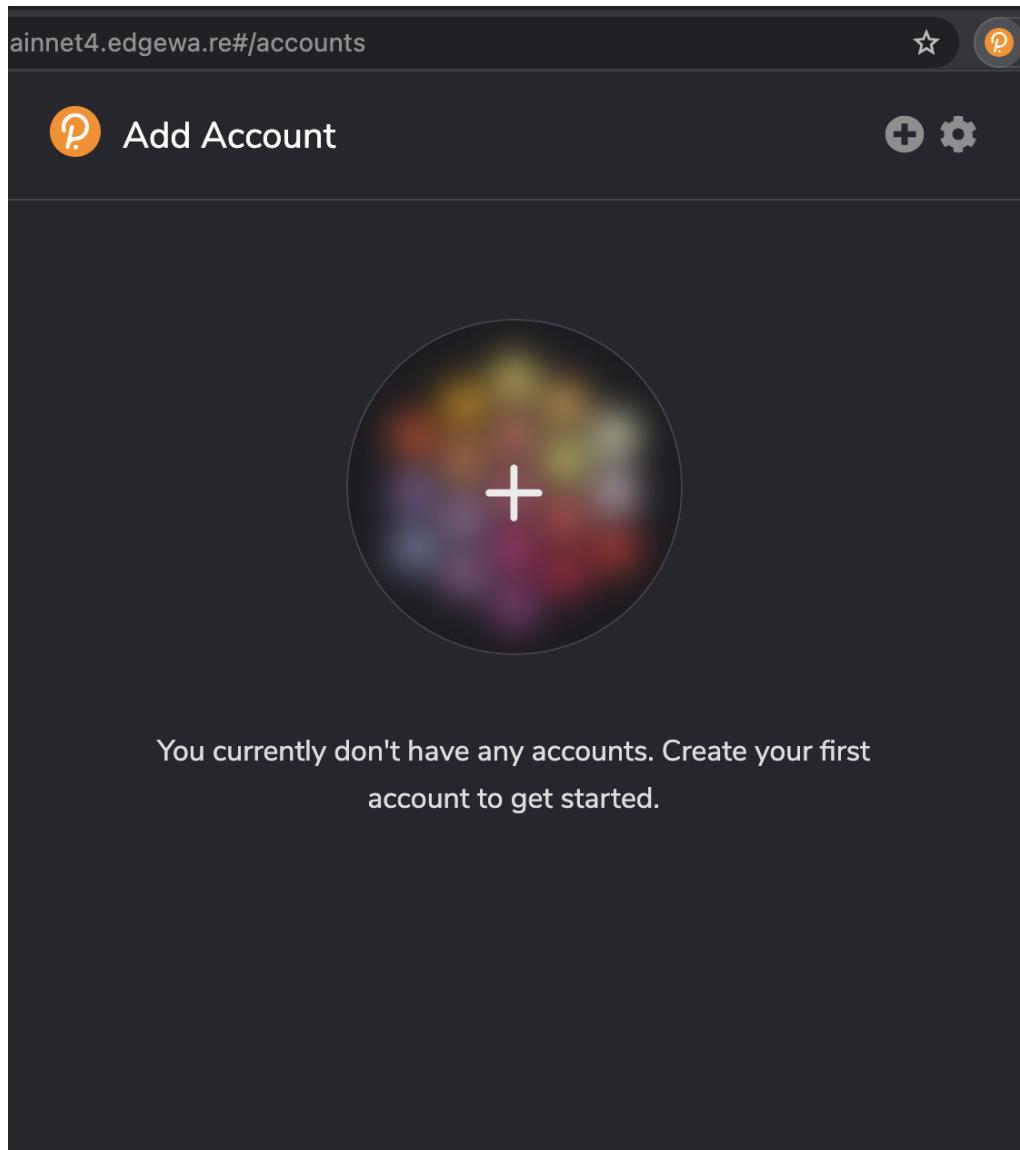
Install Polkadot Chrome Extension

Open Accounts

Navigate to [DotApps](#). Click on the "Accounts" tab.

Create Account

Open the Polkadot{.js} browser extension by clicking the logo on the top bar of your browser. You will see a browser popup not unlike the one below.



[create account in polkadot extension](#)

Click the big plus button or select "Create new account" from the small plus icon in the top right. The Polkadot{.js} plugin will then use system randomness to make a new seed for you and display it to you in the form of twelve words.



Create an account 1/2

[Cancel](#)

<unknown>

n99iXXsVzFxixAV1ySykgA5WHkSVyb1k2HzDdMtGdN1ZPxX



GENERATED 12-WORD MNEMONIC SEED:

zone curve walnut lawn nuclear cabbage hungry coconut close
domain prevent stereo

[Copy to clipboard](#)



Please write down your wallet's mnemonic seed and keep it in a safe place. The mnemonic can be used to restore your wallet. Keep it carefully to not lose your assets.



I have saved my mnemonic seed safely.

Next step

mnemonic seed for new account

You should back up these words as explained above. It is imperative to store the seed somewhere safe, secret, and secure. If you cannot access your account via Polkadot{.js} for some reason, you can re-enter your seed through the "Add account menu" by selecting "Import account from pre-existing seed".

Import account



<unknown>
<unknown>



EXISTING 12 OR 24-WORD MNEMONIC SEED

[import account to extension](#)

Name Account

The account name is arbitrary and for your use only. It is not stored on the blockchain and will not be visible to other users who look at your address via a block explorer. If you're juggling multiple accounts, it helps to make this as descriptive and detailed as needed.

Enter Password

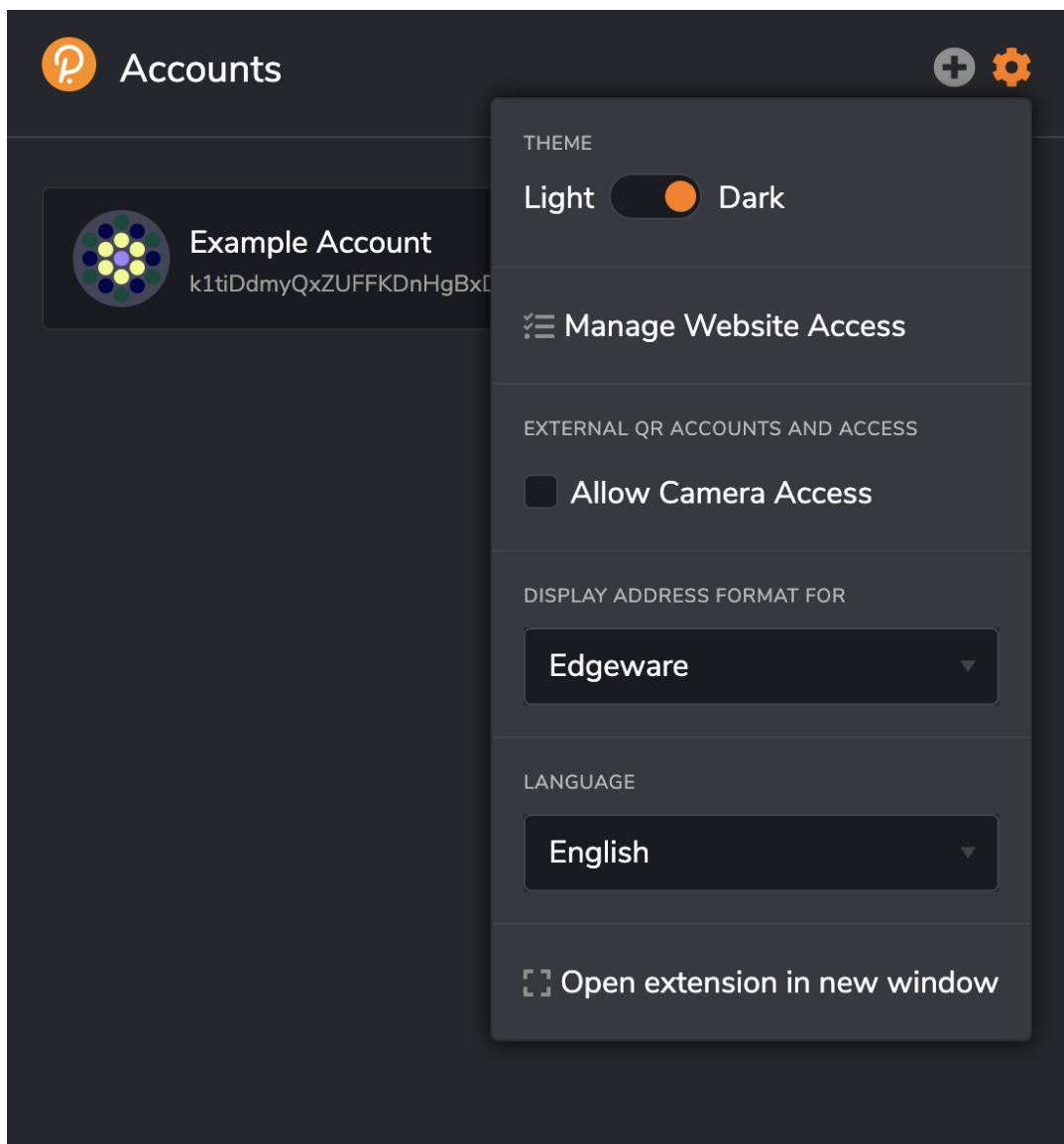
The password will be used to encrypt this account's information. You will need to re-enter it when using the account for any kind of outgoing transaction or when using it to cryptographically sign a message.

Note that this password does NOT protect your seed phrase. If someone knows the twelve words in your mnemonic seed, they still have control over your account even if they do not know the password.

Set Address for Edgeware Mainnet

Now we will ensure that the addresses are displayed as Edgeware mainnet addresses.

Click on "Options" at the top of the plugin window, and under "Display address format for" select "Edgeware".



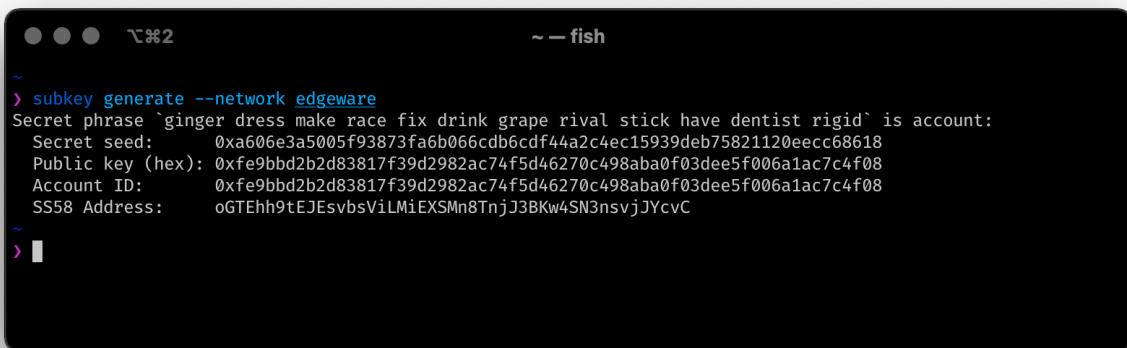
Set Address for Edgeware Mainnet

Your address' format is only visual - the data used to derive this representation of your address are the same, so you can use the same address on multiple chains. However, for privacy reasons, we recommend creating a new address for each chain you're using.

Subkey

Subkey is recommended for technically advanced users who are comfortable with the command line and compiling Rust code. Subkey allows you to generate keys on any device that can compile the code. Subkey may also be useful for automated account generation using an air-gapped device. It is not recommended for general users.

You can find detailed build and usage instructions of subkey

A screenshot of a terminal window titled "fish". The command "subkey generate --network edgeware" is run, followed by a secret phrase. The output shows the secret seed (hex), public key (hex), account ID, and SS58 address. The terminal has a dark background with white text and a black cursor.

```
~ └── fish
~ > subkey generate --network edgeware
Secret phrase `ginger dress make race fix drink grape rival stick have dentist rigid` is account:
Secret seed: 0xa606e3a5005f93873fa6b066cdb6cdf44a2c4ec15939deb75821120eecc68618
Public key (hex): 0xfe9bbd2b2d83817f39d2982ac74f5d46270c498aba0f03dee5f006a1ac7c4f08
Account ID: 0xfe9bbd2b2d83817f39d2982ac74f5d46270c498aba0f03dee5f006a1ac7c4f08
SS58 Address: oGTEhh9tEJEsvbsViLMiEXSMn8TnjJ3BKw4SN3nsvjJYcvC
~ > █
```

subkey generate address for edgeware

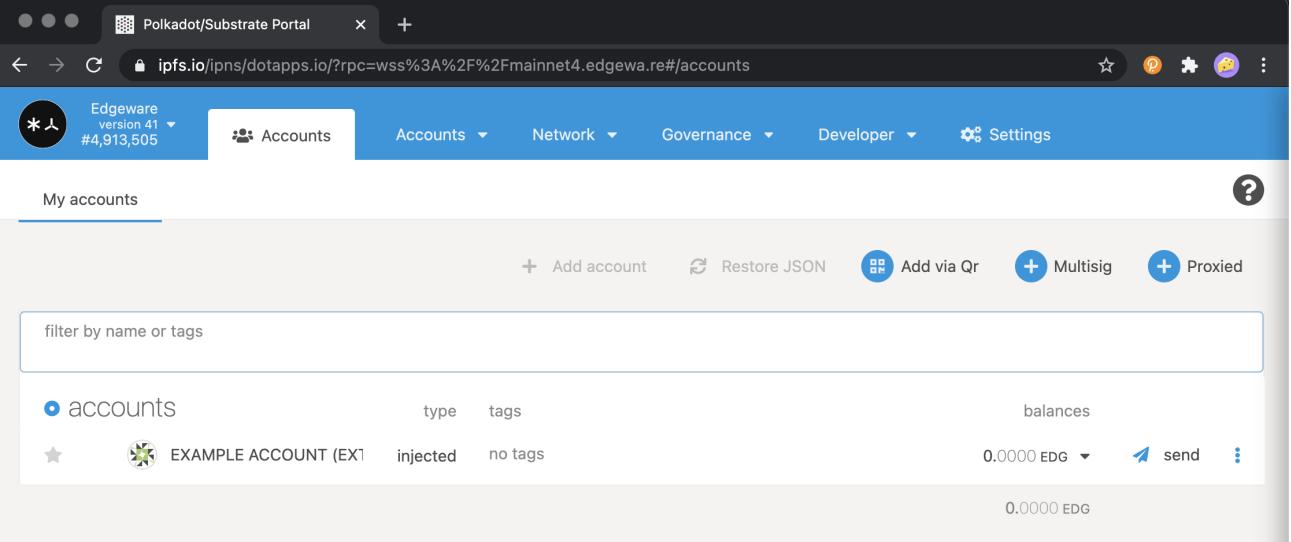
DotApps.io

Please note! If you use this method to create your account and clear your cookies in your browser, your account will be lost forever if you do not back it up. Make sure you store your seed phrase in a safe place, or download the account's JSON file if using the Polkadot{.js} browser extension. Learn more about account backup and restoration [here](#).

Using the DotApps.io user interface without the plugin is not recommended. It is the least secure way of generating an account. It should only be used if all of the other methods are not feasible in your situation.

Open DotApps.io

Navigate to [DotApps](#) and click on "Accounts" underneath the Accounts tab. It is located in the navigation bar at the top of your screen.



The screenshot shows the Polkadot/Substrate Portal interface. At the top, there's a header with the title 'Polkadot/Substrate Portal' and a URL 'ipfs.io/ipns/dotapps.io?rpc=wss%3A%2F%2Fmainnet4.edgeware.re#/accounts'. Below the header is a navigation bar with tabs: 'Accounts', 'Network', 'Governance', 'Developer', and 'Settings'. On the left, there's a sidebar with 'Edgeware version 41' and '#4,913,505'. The main content area is titled 'My accounts' and contains a table with one row. The table columns are 'accounts', 'type', 'tags', and 'balances'. The single account listed is 'EXAMPLE ACCOUNT (EX1)' which is 'injected' and has 'no tags'. The 'balances' column shows '0.0000 EDG' and '0.0000 EDG'. There are buttons for 'Add account', 'Restore JSON', 'Add via QR', 'Multisig', and 'Proxied'. A search bar at the top says 'filter by name or tags'.

DotApps Accounts tab

Start Account Generation

Click on the "Add Account" button. You should see a pop-up similar to the process encountered when using the [Polkadot JS Extension method](#) above. Follow the same instructions and remember to [store your seed safely!](#)

Create and Back Up Account

Click "Save" and your account will be created. It will also generate a backup JSON file that you should safely store, ideally on a USB off the computer you're using. You should not store it in cloud storage, email it to yourself, etc.

You can use this backup file to restore your account. This backup file is not readable unless it is decrypted with the password.

Reference

- Account generation for Polkadot Relay chain
- SS58 Address Transform

Connect to a Wallet and Check Balance

Follow this guide to connect your Edgeware address to a wallet and get started on the network by first checking your balance. Please note, if you received your EDG from an exchange, you won't need to follow the step of "converting a Lockdrop Address" in this guide and instead can directly connect your account in step two.

Convert a Lockdrop Address to the New Format

Lockdrop addresses have an outdated format and need to be transformed to find your accounts on block explorers like Subscan.io. Subscan has a great tool for this, you can enter your public address to see the full list of network transformations of your address. You'll want to copy the one for Edgeware. Scroll down for background or more ways to do this.

https://edgeware.subscan.io/tools/ss58_transform

The screenshot shows the Subscan.io website with the URL https://edgeware.subscan.io/tools/ss58_transform. The page title is "ss58 Address Transform". On the left, there is an input field labeled "Input Account or Public Key" containing the lockdrop address: 0x1a61591442ffb7bd0bfac278533aac01c1ca a0335d65c0af415bc912d7d3df03. Below this is a "Transform" button. An arrow points to the right, leading to a list of network transformations. The list includes:

- Darwinia (Prefix: 18)
2omTpLPJheUL4tsGWUfZabvtpkvF0TrmwRk77FHfAacRqGD6
- Crab (Prefix: 42)
5CIJ29NyvdTmpJaQFFn8eEK6LQMYzCaGU13FseitH6t8aoA5
- Westend (Prefix: 42)
5CIJ29NyvdTmpJaQFFn8eEK6LQMYzCaGU13FseitH6t8aoA5
- Edgeware (Prefix: 7)
i7CyRRsUsrLMFRAYczJRh1DirMG4nY3s5efLjDM5B54X37
- Mandala (Prefix: 42)
5CIJ29NyvdTmpJaQFFn8eEK6LQMYzCaGU13FseitH6t8aoA5

Each transformation entry has a "Copy" icon to its right.

Background

During the lockdrop, the keypairs that were generated with Subkey were encoded using the Subkey Default Network ID. This impacts the Public Address (SS58 Address Format) that Subkey outputs in certain cases. The secret phrase/seed, and public key are not impacted by this change.

1. You can always use the Default-Network-ID-encoded Public Address to **receive** funds safely.
2. However, It is a 'best practice' to use the _new Edgeware-_network-ID version of your address when interacting with the Edgeware network, including sending funds.
3. You will need to re-generate your public address to use some Block Explorer tools. At this time, Polkascan uses the new Edgeware network ID encoding.

You can also get the new version of your address in two other ways - the Polkadot UI or the Subkey tool.



The Polkadot.js **Browser Extension** does not display the Edgeware network ID encoded at this time, **but the Polkadot UI does**.

Easy Mode: Subscan Tool

Visit this [Subscan Tool](#) and enter your address or public key to generate a list of many network-encoded versions of your address. Save the one marked Edgeware.

The screenshot shows the Subscan Tool's interface. At the top, it displays "SUBSCAN EDG \$0.009 (-1.62%)". The navigation bar includes Home, Blockchain, Accounts, Governance, Tools, 中文, and a search bar with the query "* Edgeware". Below the navigation, there is a dropdown menu set to "All" and a search input field. A "Search" button is located to the right of the search bar. The main content area is titled "ss58 Address Transform". On the left, a panel titled "Input Account or Public Key" contains a text input field with the value "0x1a61591442ff7bd0bfa...". To the right of this input field is a circular icon with a green arrow. Below the input field is a "Transform" button. An arrow points from this panel to a list of five network addresses on the right. The list includes:

- Darwinia (Prefix: 18)
2omTpLJheUL4tsGWUfZabvtpkvVfoTmwRk77FHfAacRqGD6
- Crab (Prefix: 42)
5CfJ29NyvdTmpJaQFFn8eEK6LQMYzCaGU13FseitH6t8aoA5
- Westend (Prefix: 42)
5CfJ29NyvdTmpJaQFFn8eEK6LQMYzCaGU13FseitH6t8aoA5
- Edgeware (Prefix: 7)
i7CyRRsUsrLMFRAYczJRh1DiRMG4nY3s5eIJDM5B54jX37
- Mandala (Prefix: 42)
5CfJ29NyvdTmpJaQFFn8eEK6LQMYzCaGU13FseitH6t8aoA5

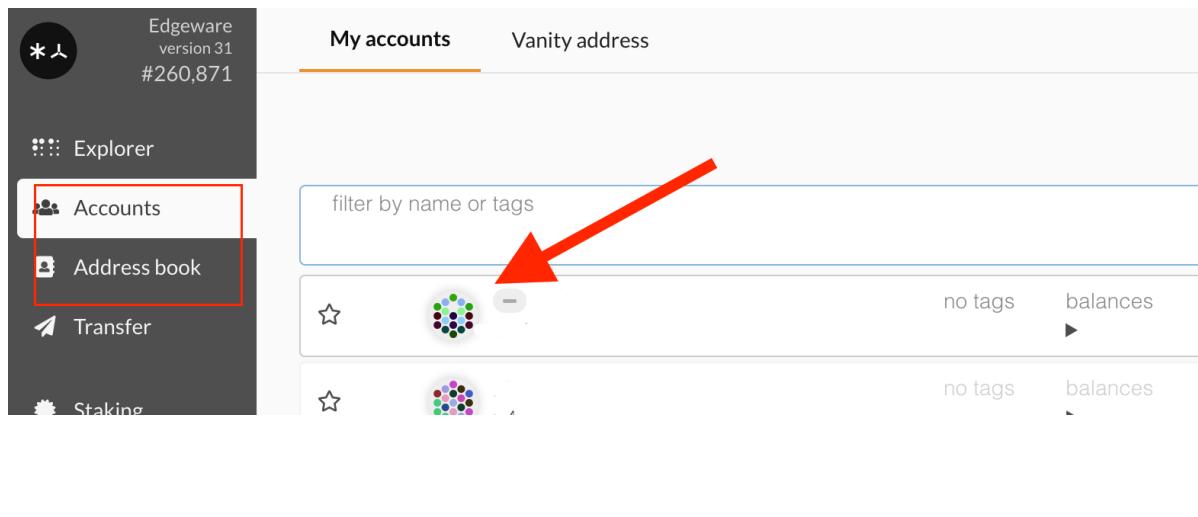
Each item in the list has a small circular icon with a green arrow to its right. A "Copy All" button is located at the top right of the list area.

Using Polkadot UI

The easiest way to get your Edgeware network ID version of your public address is to use the Account or Address Book tool of [the Polkadot UI](#).

Enter your old address into the search bar in the Address Book tool, and then click the identicon to the left as shown in the image below. It will copy your current Edgeware network encoded address. Store this safely.

Alternatively, you can use the Accounts tab to do the same if you've connected via the Polkadot UI Browser Extension.



-  This process requires the latest version of the Subkey program in order to select Edgeware as the network.

You can generate a new key with the Edgeware network ID using

```
subkey -n edgeware generate
```

You can also derive the SS58 Address from **an existing key OR address** with the Edgeware Network ID by using

```
subkey -n edgeware inspect "INSERT MNEMONIC/EXISTING ADDRESS HERE"
```

Examples

In order of examples:

1. Generate a new key with `subkey -n edgeware`
2. Inspect key with default network ID (as you can see the SS58 is different)
3. Inspect key with **edgeware network ID** (as you can see it matches the first example entirely)

```
1 → subkey -n edgeware generate
2
3 Secret phrase `submit hotel naive plate among decorate ghost speak ex
4   Secret seed:      0xbe59988b1ad6e93325766b9ca8713db866538f75b578f71
5   Public key (hex): 0x0045d45397c474d6a4fddb00e314637c8970be097f5a9a2
```

```
6 Account ID: 0x0045d45397c474d6a4fddb00e314637c8970be097f5a9a2
7 SS58 Address: hWyz5UkvkrPpmiD3MJ7htdhEigxF34pUH3Dy9Pq71QeZg4
8
9 → subkey inspect "submit hotel naive plate among decorate ghost spe
10
11 Secret phrase `submit hotel naive plate among decorate ghost speak ex
12 Secret seed: 0xbe59988b1ad6e93325766b9ca8713db866538f75b578f71
13 Public key (hex): 0x0045d45397c474d6a4fddb00e314637c8970be097f5a9a2
14 Account ID: 0x0045d45397c474d6a4fddb00e314637c8970be097f5a9a2
15 SS58 Address: 5C54boRsNwTqHpsSjz5wvRwZrhhFB45HRPfdktew32pUW8CZ
16
17 → subkey -n edgeware inspect "submit hotel naive plate among decorat
18
19 Secret phrase `submit hotel naive plate among decorate ghost speak ex
20 Secret seed: 0xbe59988b1ad6e93325766b9ca8713db866538f75b578f71
21 Public key (hex): 0x0045d45397c474d6a4fddb00e314637c8970be097f5a9a2
22 Account ID: 0x0045d45397c474d6a4fddb00e314637c8970be097f5a9a2
23 SS58 Address: hWyz5UkvkrPpmiD3MJ7htdhEigxF34pUH3Dy9Pq71QeZg4
```



- On Commonwealth.im, when Edgeware mainnet is enabled, the inputs will auto-derive the SS58 with the correct network ID, however, using the default-ID-ss58 outside of Commonwealth.im will likely result in errors that may endanger funds. To be safe, always use the Edgeware Network ID SS58 Address.

Connect Your Account

There are several ways to connect your existing account from the Lockdrop or other events to an account manager or wallet.

First, install the [Polkadot.js Browser Extension](#):

Polkadot.js Extension

- On Chrome, install via [Chrome web store](#)
- On Firefox, install via [Firefox add-ons](#)
- You will need your seed phrase, a series of words.

Next, click the Orange P icon in your browser extension section.

Click the button stating "I have a pre-existing seed, import the account."

Enter your mnemonic phrase and hit the confirm button.

Next, visit <https://polkadot.js.org/apps/#/explorer>, ensure you are connected to [the Edgeware network](#) by clicking the top left network logo and selecting the network you want to connect to.

Once connected, the extension will prompt you to authorize connecting your local wallet to the service. Select your account, enter your wallet password and authorize the interaction.

You can now explore the chain and your account on Edgeware.

IMPORT ACCOUNT

[Back](#)

existing 12 or 24-word mnemonic seed
eager rocket year stamp behave review shiver feed intact
polar end waste

a descriptive name for this account
My Edgeware Account

a new password for this account

repeat password for verification



My Edgeware Account

5ECT3M6MuYdUAwCvAPMSJQqzvRLZQ8BbjYv...

Add the account with the supplied seed

Commonwealth.im

Finally, enter in a name and password twice for your newly connected account. Once connected, you can visit <https://polkadot.js.org/apps/#/explorer>, connect to the Edgeware testnet, and view your balance and interact with the chain.

Check your Balance via Block Explorer

At this time, the two main block explorers are

- [Polkascan](#)

- Subscan

Both of these explorers will only work with Public Addresses (SS58) that have the Edgeware network ID encoded.

-  If you have not re-encoded your public address from your Lockdrop-created keypair with the new Edgeware network ID, you **must do so** in order to use block explorers to find your account. Follow the instructions linked below.

Once you have your re-encoded Public Address, you can enter it into the search bar in a block explorer to pull up your account and view the balance.

Send EDG to another Account

Using Polkadot.js

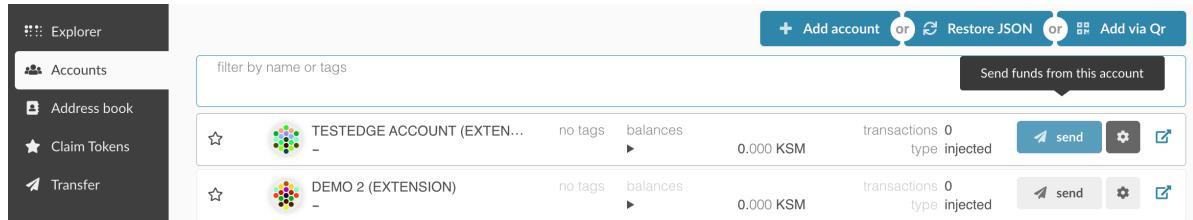
This method involves using the [Polkadot UI](#) and the Polkadot.js browser extension.

You will need:

- ..to have [created an Edgeware account](#).
- ..to have installed [Polkadot.js](#) to your browser and connected your wallet.
- Some testnet or mainnet EDG to send.

Visit the [Polkadot UI](#) and ensure you are connected to the Edgeware network you want to participate in by clicking the network section in the top left. Once connected, click the Accounts tab in the left sidebar, or going directly to the Transfer tab.

In the Accounts tab, click the send button in the row of the account you want to send from.



The screenshot shows the Polkadot UI interface. On the left, there's a sidebar with options: Explorer, Accounts (which is selected), Address book, Claim Tokens, and Transfer. The main area is titled 'Accounts' and has a search bar labeled 'filter by name or tags'. Below the search bar, there are two account entries:

Account	Type	Balance	Transactions	Actions
TESTEDGE ACCOUNT (EXTENSION)	-	0.000 KSM	0	Send Settings Copy
DEMO 2 (EXTENSION)	-	0.000 KSM	0	Send Settings Copy

Next, confirm the accounts you want to send from and send to, using the address fields. You can delete the information in the 'send to address' to enter in any recipient address, or utilize the dropdown to send between your own connected accounts in your Polkadot.js wallet.

Once the amount is entered and all the information is reviewed and confirmed (treat cryptocurrency transactions as irreversible, so be careful,) click "Make Transfer."

Send funds

The screenshot shows the 'Send funds' interface. It has two sections for transfers:

- send from account**: TESTEDGE ACCOUNT (EXTENSION) with address DMwujhDrTNXKuMJEBiLT3rcadhV63m5J6DpRKQ9G2cwu... and balance transferrable 0.000 KSM.
- send to address**: DEMO ACCOUNT (EXTENSION) with address HLQLZci2XvbKZWaFjNXAdpE18weSGoboDSY7BRB4jCoG... and balance transferrable 0.000 KSM.

Below these is a field for 'amount' with value 0 and a dropdown for currency set to KSM. At the bottom are 'Cancel' and 'Make Transfer' buttons.

You will see a screen similar to this next, where you can see your transaction fees, can include a tip to the validator who authors the block for faster processing of the transaction (uncommon,) and otherwise confirm the send by signing the transaction.

Advanced: You can also pre-sign but *not* submit this transaction to the network (uncommon) by using the bottom left 'Sign and Submit' toggle and entering a nonce and a duration for the validity of the signed transaction.

The screenshot shows the transaction submission interface. At the top, it says 'balances.transfer' and 'queued'. It details a transfer from TESTEDGE ACCOUNT (EXTENSION) to DEMO ACCOUNT (EXTENSION) with value 0 KSM. A note says 'Fees of 10.000 milli KSM will be applied to the submission'. Below this is a 'Sign and Submit' toggle switch, which is turned off. At the bottom are 'Cancel' and 'Sign and Submit' buttons.

Once you hit sign and submit, your Polkadot.js browser extension will open a popup for your account password for the 'send from' account, and you will sign the transaction from your wallet.

TRANSACTIONS



TestEDGE Account

Kusama CC3

DMwujhDrTNXKuMJEBiLT3rcadhV63m5J6DpRK...

from <https://polkadot.js.org/apps/#/settings>

chain Kusama CC3

version 1045

nonce 0

method data 0x0400ffd28ff54e25d9a0943ff479f460006ca151515f...

lifetime mortal, valid from #989,188 to #989,252

password for this account

Sign the transaction

Cancel

Once you sign the transaction, the network receives it and you are done. You can explore the transaction details through a block explorer.

Using Commonwealth.im

Coming soon.

Create an Edgeware Identity

Identity

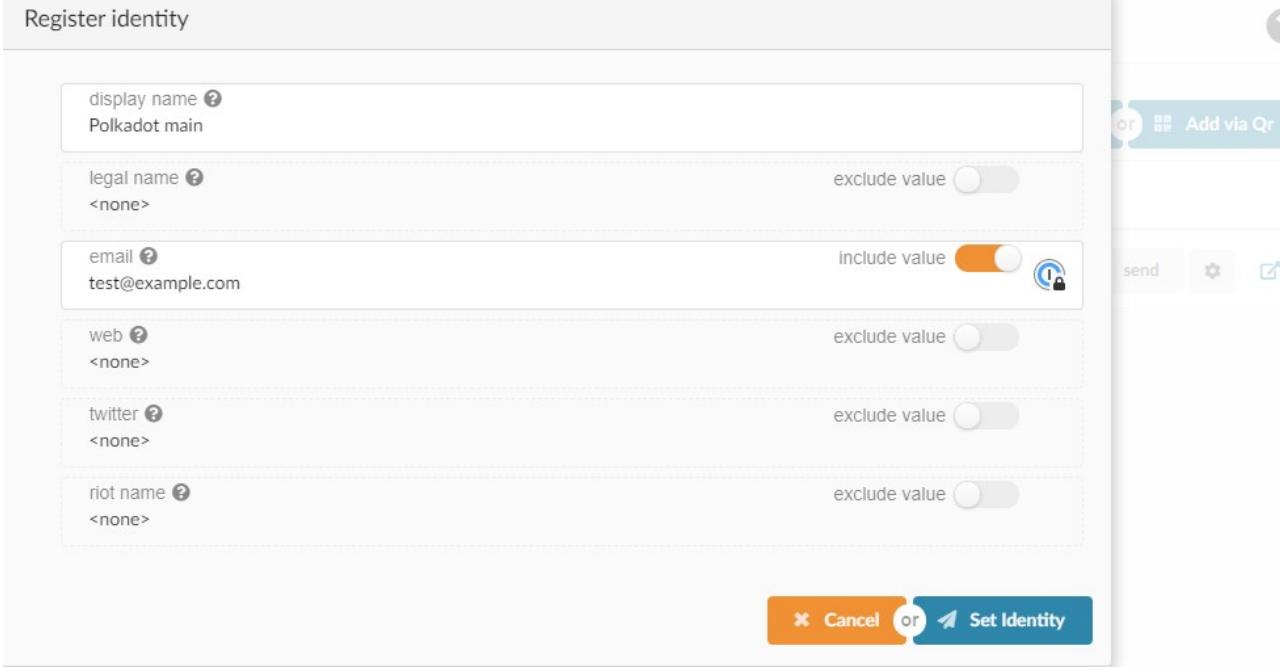
Substrate provides a naming system that allows participants to add personal information to their on-chain account and subsequently ask for verification of this information by registrars.

[Learn more here](#)

Adding Identities Using Polkadot.js

The easiest way to add the built-in fields is to click the gear icon next to one's account and select "Set on-chain identity".

The screenshot shows the Polkadot.js web interface. At the top, there are tabs for 'My accounts' (which is selected) and 'Vanity address'. Below the tabs is a search bar with the placeholder 'filter by name or tags' and a dropdown menu showing 'main'. On the right side of the header are buttons for '+ Add account', 'Restore JSON', and 'Add via QR'. The main content area displays an account card for 'POLKADOT MAIN'. The card includes a star icon, a colorful hexagonal icon, the account name 'POLKADOT MAIN', a 'no tags' label, a 'balances' button, a balance of '0.000 KSM', transaction details ('0 transactions', 'type sr25519'), and buttons for 'send', 'gear icon', and 'copy'. A context menu is open over the account name, listing options: 'Set on-chain identity' (which is bolded), 'Derive account via derivation path', 'Create a backup file for this account', 'Change this account's password', 'Forget this account', 'Make recoverable', 'Initiate recovery for another', and a toggle switch for 'use on any network'.



To add custom fields beyond the default ones, use the Extrinsic UI to submit a raw transaction by first clicking "Add Item" and adding any field name you like. The example below adds a field `steam` which is a user's **Steam** username. The first value is the field name in bytes ("steam") and the second is the account name in bytes ("theswader"). The display name also has to be provided, otherwise the Identity pallet would consider it wiped if we submitted it with the "None" option still selected. That is to say, every time you make a change to your identity values, you need to re-submit the entire set of fields: the write operation is always "overwrite", never "append".

The rendering of such custom values is, ultimately, up to the UI/dapp makers. In the case of PolkadotJS, the team prefers to only show official fields for now. If you want to check that the values are still stored, use the **Chain State UI** to query the active account's identity info:

```

identity.identityOf: Option<Registration>
{"judgements":[],"deposit":1250000000000,"info":{"additional":[{"Raw":"0x737465616d"}, {"Raw":"0x746865737761646572"}],"display":{"Raw":"0x69642d74657374"}, "legal": {"None":null}, "web": {"None":null}, "riot": {"None":null}, "email": {"None":null}, "pgpFingerprint":null, "image": {"None":null}, "twitter": {"None":null}}}
  
```

Raw values of custom fields are available on-chain

It is up to your own UI or dapp to then do with this data as it pleases. The data will remain available for querying via the Polkadot API, so you don't have to rely on the PolkadotJS UI.

You can have a maximum of 100 custom fields.

Format Caveat

Please note the following caveat: because the fields support different formats, from raw bytes to various hashes, a UI has no way of telling how to encode a given field it encounters. The PolkadotJS UI currently encodes the raw bytes it encounters as UTF8 strings, which makes these values readable on screen. However, given that there are no restrictions on the values that can be placed into these fields, a different UI may interpret them as, for example, IPFS hashes or encoded bitmaps. This means any field stored as raw bytes will become unreadable by that specific UI. As field standards crystallize, things will become easier to use but for now, every custom implementation of displaying user information will likely have to make a conscious decision on the approach to take, or support multiple formats and then attempt multiple encodings until the output makes sense.

Set Up a Full Node

This guide covers how to set up an Edgeware node. There are two ways you can proceed:

- Setting up a private node, e.g. if you would like to run a validator
- Setting up a public node, e.g. if you want to run connect services or dapps to Edgeware

 To quickly run node from **Docker** image:

<https://hub.docker.com/repository/docker/hicommonwealth/edgeware>

If you are running a private node, you will only need to follow **steps 0 and 1** of this guide. Otherwise, we will guide you through setting up an SSL certificate in **steps 2 and 3**, so any browser can securely connect to your node. (Most people, including validators, only need to set up a private node.)

0. Provisioning a server

Provision an appropriately sized server from a reputable VPS provider, e.g.:

- [Vultr](#)
- [DigitalOcean](#)
- [Linode](#)
- [OVH](#)
- [Contabo](#)
- [Scaleway](#)
- Amazon AWS, etc.

We recommend a node with at least 2GB of RAM, and Ubuntu 18.04 x64. Other operating systems will require adjustments to these instructions.

If you are running a public node, set up DNS from a domain name that you own to point to the server. We will use `testnet1.edgewa.re`. (You don't need to do this if you are setting up a private node.)

SSH into the server.

1. Installing Edgeware and setting it up as a system service

First, clone the `edgeware-node` repo, install any dependencies, and run the required build scripts.

```
1 apt update
2 apt install -y gcc libc6-dev
3 apt install -y cmake pkg-config libssl-dev git clang libclang-dev
4
5 # Prefetch SSH publickeys
6 ssh-keyscan -H github.com >> ~/.ssh/known_hosts
7
8 # Install rustup
9 curl https://sh.rustup.rs -sSf | sh -s -- -y
10 source /root/.cargo/env
11 export PATH=/root/.cargo/bin:$PATH
12
13 # Get packages
14 git clone https://github.com/hicommonwealth/edgeware-node.git
15 cd edgeware-node
16
17 # Build packages
18 ./setup.sh
```

Set up the node as a system service. To do this, navigate into the root directory of the `edgeware-node` repo and execute the following to create the service configuration file:

```
1 {
2     echo '[Unit]'
3     echo 'Description=Edgeware'
4     echo '[Service]'
5     echo 'Type=exec'
6     echo 'WorkingDirectory='`pwd`'
7     echo 'ExecStart='`pwd`'/target/release/edgeware --chain=edgeware --ws-e
8     echo '[Install]'
9     echo 'WantedBy=multi-user.target'
10 } > /etc/systemd/system/edgeware.service
```

Note: This will create an Edgeware server that accepts incoming connections from anyone on the internet. If you are using the node as a validator, you should instead remove the `ws-external` flag, so Edgeware does not accept outside connections.

Double check that the config has been written to `/etc/systemd/system/edgeware.service` correctly. If so, enable the service so it runs on startup, and then try to start it now:

```
1 systemctl enable edgeware  
2 systemctl start edgeware
```

Check the status of the service:

```
systemctl status edgeware
```

You should see the node connecting to the network and syncing the latest blocks. If you need to tail the latest output, you can use:

```
journalctl -u edgeware.service -f
```

2. Configuring an SSL certificate (public nodes only)

We will use Certbot to talk to Let's Encrypt. Install Certbot dependencies:

```
1 apt -y install software-properties-common  
2 add-apt-repository universe  
3 add-apt-repository ppa:certbot/certbot  
4 apt update
```

Install Certbot:

```
apt -y install certbot python-certbot-nginx
```

It will guide you through getting a certificate from Let's Encrypt:

```
certbot certonly --standalone
```

If you already have a web server running (e.g. nginx, Apache, etc.) you will need to stop it, by running e.g. `service nginx stop`, for this to work.

Certbot will ask you some questions, start its own web server, and talk to Let's Encrypt to issue a certificate. In the end, you should see output that looks like this:

```
1 root:~/edgeware-node# certbot certonly --standalone
2 Saving debug log to /var/log/letsencrypt/letsencrypt.log
3 Plugins selected: Authenticator standalone, Installer None
4 Please enter in your domain name(s) (comma and/or space separated) (Enter
5 to cancel): testnet1.edgewa.re
6 Obtaining a new certificate
7 Performing the following challenges:
8 http-01 challenge for testnet1.edgewa.re
9 Waiting for verification...
10 Cleaning up challenges
11
12 IMPORTANT NOTES:
13 - Congratulations! Your certificate and chain have been saved at:
14   /etc/letsencrypt/live/testnet1.edgewa.re/fullchain.pem
15   Your key file has been saved at:
16   /etc/letsencrypt/live/testnet1.edgewa.re/privkey.pem
17   Your cert will expire on 2019-10-08. To obtain a new or tweaked
18   version of this certificate in the future, simply run certbot
19   again. To non-interactively renew *all* of your certificates, run
20   "certbot renew"
```

3. Configuring a Websockets proxy (public nodes only)

First, install nginx:

```
apt -y install nginx
```

Set the intended public address of the server, e.g. `testnet1.edgewa.re`, as an environment variable:

```
export name=testnet1.edgewa.re
```

Set up an nginx configuration. This will inject the public address you have just defined.

```
1  {
2    echo 'user      www-data; ## Default: nobody'
3    echo 'worker_processes 5; ## Default: 1'
4    echo 'error_log  /var/log/nginx/error.log;'
5    echo 'pid      /var/run/nginx.pid;'
6    echo 'worker_rlimit_nofile 8192;'
7    echo ''
8    echo 'events {'
9    echo '  worker_connections 4096; ## Default: 1024'
10   echo '}'
11   echo ''
12   echo 'http {'
13   echo '    map $http_upgrade $connection_upgrade {'
14   echo '        default upgrade;'
15   echo "        \\" close;"'
16   echo '}'
17   echo '    server {'
18   echo '        listen      443 ssl;'
19   echo '        server_name '$name';'
20   echo ''
21   echo '        ssl_certificate /etc/letsencrypt/live/'$name'/cert.pem;'
22   echo '        ssl_certificate_key /etc/letsencrypt/live/'$name'/privkey.pem;'
23   echo '        ssl_session_timeout 5m;'
24   echo '        ssl_protocols  SSLv2 SSLv3 TLSv1;'
25   echo '        ssl_ciphers  HIGH:!aNULL:!MD5;'
26   echo '        ssl_prefer_server_ciphers  on;'
```

```
27     echo ''  
28     echo '         location / {'  
29     echo '             proxy_pass http://127.0.0.1:9944 ;'  
30     echo '             proxy_http_version 1.1;'  
31     echo '             proxy_set_header Upgrade $http_upgrade;'  
32     echo '             proxy_set_header Connection $connection_upgrade;'  
33     echo '         }'  
34     echo '     }'  
35     echo '}'  
36 } > /etc/nginx/nginx.conf
```

Make sure that the paths of `ssl_certificate` and `ssl_certificate_key` match what Let's Encrypt produced earlier. Check that the configuration file has been created correctly.

```
1 cat /etc/nginx/nginx.conf  
2 nginx -t
```

If there is an error, `nginx -t` should tell you where it is. **Note that there may be subtle variations in how different systems are configured, e.g. some boxes may have different login users or locations for log files. It is up to you to reconcile these differences.**

Start the server:

```
service nginx restart
```

You can now try to connect to your new node from [polkadot.js/apps](#), or by making a curl request that emulates opening a secure WebSockets connection:

```
curl --include --no-buffer --header "Connection: Upgrade" --header "Upgrade:
```

4. Connecting to your node

Congratulations on your new node! If you set up public DNS and a SSL certificate in steps 2 and 3, you should be able to connect to it now from [polkadot.js/apps](#):



Otherwise, you should be able to use [edgeware-cli](#) to connect to it:

```
1 git clone https://github.com/hicommonwealth/edgeware-cli.git
2 cd edgeware-cli
3 yarn
4 bin/edge -r ws://testnet1.edgewa.re:9944 balances freeBalance 5G8jA2TLTQqno
```

In general, you should use these URLs to connect to your node:

- `ws://testnet1.edgewa.re:9944` if you set it up as a public node with `--ws-external` in step 1
- `wss://testnet1.edgewa.re` if you set it up as a public node and also followed steps 2 and 3

5. Next steps

Your node will automatically restart when the system reboots, but it may not be able to recover from other failures. To handle those, consider following our guide to [Setting up monitoring](#).

You may also wish to proceed to [Validating on Edgeware](#).

Set Up a Validator

Welcome to the official, in-depth Edgeware guide to validating. We're happy that you're interested in validating on Edgeware and we'll do our best to provide in-depth documentation on the process below. As always, reach out on [Discord](#) or [Telegram](#) if you have questions about the project.

This document contains all the information one should need to start validating on Edgeware using the **command line interface**. We will start with how to setup one's node and proceed to how to key management and monitoring. To start, we will use the following terminology of keys for the guide:

- **stash** - the stash keypair is where most of your funds should be located. It can be kept in cold storage if necessary.
 - **controller** - the controller is the keypair that will control your validator settings. It should have a smaller balance, e.g. 10-100 EDG
 - **session** - the 4 session keypairs are hot keys that are stored on your validator node. They do not need to have balances.
-

Requirements

1. You will need 6 keypairs: a `stash` (ed25519 or sr25519), `controller` (ed25519 or sr25519), and 4 `session` (3 ed25519 and 1 sr25519) keypairs. You can generate these using the `subkey` utility. We will be using derived keys in the examples, if you do not use derived keys, simply input the seed/mnemonic needed to sign from these accounts.
2. Aura keys (ed25519)
3. Grandpa keys (ed25519)
4. ImOnline keys (ed25519)
5. AuthorityDiscovery keys (sr25519)
6. You will need at least the existential balance (1,000,000,000,000,000 token units i.e 0.0001 EDG) in both the `stash` and `controller` accounts plus the balances needed to send transactions from these accounts.
7. You will need a live, fully-synced Edgeware node running with the `--validator` flag that has set one's session keys, either before or after you complete the onboarding

process.

Pre-requisites

- First follow the guide in the [README.md](#) for installing and running the `edgeware-node`.
- Download from source or from the `npm` registry the `edgeware-cli` located [here](#).
Note: `edgeware-cli` has several dependencies [viewable here](#).
- Install `subkey` as well if you do need to generate new keypairs:
`cargo install --force --git https://github.com/paritytech/substrate subkey`

From this point on, we will assume you are familiar with using `subkey`, if that is not the case, you can read about the `subkey` commands [here](#).

Onboarding

1. First, create the **stash** and **controller** keypairs using `subkey`. You can also **optionally** **create your 4 session keys. Create ED25519 keypairs using `-e` flag with subkey.
2. Next, you will need to bond from your **stash** keypair to your **controller** keypair. Using the CLI and a local node, you will run:

```
edge -s <STASH_SEED> staking bond <CONTROLLER_B58_ADDRESS> <AMOUNT> <REWARD_DESTINATION>
```

3. The **stash** seed should be a mnemonic + derivation path for your **stash** keypair
4. The **controller** address should be a Base58 encoded public key (starts with a 5)
5. The bond **amount** should be an integer balance in units of EDG
6. The **reward destination** is where rewards will go; the options are `stash`, `controller`, and `staked` (where staked adds rewards to the amount staked)
7. Next, you will need to set your validator preferences from your **controller** account. Using the CLI and a local node, you will run:

```
edge -s <CONTROLLER_SEED> staking validate <COMMISSION_PERCENTAGE>
```

8. The **controller** seed should be a mnemonic + derivation path for your **controller** keypair
9. The **unstake threshold** is the number of times your node is offline before dropping out
10. The **commission percentage** is the percentage of rewards you will
11. Next, you will need to set your **session** keys from your **controller** keypair. Using the CLI and a local node, you will run:

```
edge -s <CONTROLLER_SEED> session setKeys <OUTPUT_FROM_ROTATE_KEYS> 0x
```

12. The **controller** seed should be a mnemonic + derivation path for your **controller** keypair
13. The **session** public keys should be concatenated from the output of the rotate keys rpc command.

Examples of all the commands are below:

In the following, we have downloaded and compiled `edgeware-cli` from source to yield a `/bin/edge` binary. You can use `tsc` to do so if you compile from source.

```
1 edge -s "axis service this custom because top clap sock weekend tenant vehicle"
2
3 edge -s "axis service this custom because top clap sock weekend tenant vehicle"
4
5 edge -s "axis service this custom because top clap sock weekend tenant vehicle"
```

Validating

The v099 testnet requires validators to manage 4 validating keys for the Aura, Grandpa, ImOnline, and AuthorityDiscovery modules.

1. Aura keys (ed25519)

2. Grandpa keys (ed25519)
3. ImOnline keys (ed25519)
4. AuthorityDiscovery keys (sr25519)

Now while running your full node to sync or afterward, you can start to set up your session keys for the node. The command for inserting keys and rotating keys is the same as it has been. To rotate new session keys, run the following while your node is running:

```
curl -H 'Content-Type: application/json' --data '{ "jsonrpc":"2.0", "method":
```

To insert existing session keys, you can run for each key the following command while your node is running:

```
curl -H 'Content-Type: application/json' --data '{ "jsonrpc":"2.0", "method":
```

The four key types you will enter individuals are:

- `aura` for Aura keys
- `gran` for Grandpa keys
- `imon` for ImOnline keys
- `audi` for AuthorityDiscovery keys

After running these `curl` commands, you should receive as output from `stdout` the public keys you provided (or didn't) in a JSON string. That also means the process was a success! You should now see yourself in the list of newly/pending validators to go into effect in future sessions. In the next era (up to 1 hour), if there is a slot available, your node will become an active validator.

Contribute & Engage

Community

Chat & Discussion Channels

Group Name	Link
Commonwealth Edgeware	Forum, governance, longform.
EDG Element	Technical, governance, serious.
General Telegram	Announcements, support, general chat.
Validator Telegram	Validation, announcements.
Edgeware Economics TG	Markets, trollbox, price.
Discord	discord.gg/njDnHDk
Twitter	twitter.com/HeyEdgeware
Edgeware Türkiye	t.me/EdgewareTUR
Edgeware India	t.me/EdgewareIndia
Edgeware&Wetez	Chinese lang. Contact WeChat ID kamiesheep for invite.

Edgeware Community Call (Monthly)

Monthly on the third Wednesday at 2pm EST

Recurring Zoom Link zoom.us/j/345881968

Meeting ID: 345 881 968

Description

Announcing the Edgeware Community Call! Each month the core development team will be available to talk with validators, users, developers and all enthusiasts about the roadmap, ideas, and the space.

Google Event for the Monthly Call

[View the Entire Edgeware Community Event Calendar on Google Apps](#)

[View the entire Community Calendar via any Web Browser](#)

[View the Calendar into other Apps using iCal/ ics](#)

Forums

Name	Link	Features
Commonwealth Edgeware	commonwealth.im/edgeware	Governance, Profiles, Discussion

Chat Rooms

It is the wish of Commonwealth Lab that users have access to a variety of platforms - but that bridges or bots unify the content.

Name	Type and Link	Topics	Requires Email
General	Telegram Group	General except market and price discussion.	No
Governance	Telegram Group	Governance	No
Economics	Telegram Group	Markets, price, trollbox.	No
Developers	Telegram Group	Developers, Builders, Tooling Question	No
Smart Contracts	Element	Builders help around Smart Contracts in ink!	No

General	Element	General except market and price discussion. Tech support.	No
Governance	Element	Governance	No
General	Discord Chat	General, Validation, Tech Support	Yes

Social Media and Announcements

Name	Link
News Twitter	twitter.com/HeyEdgeware
Developers Twitter	twitter.com/EDG_Developers
General Announcements TG Channel	t.me/edgeware_announcements
Validator Updates TG Channel	t.me/EdgewareValidators

Working Groups

Name	Link
Governance	commonwealth.im/edgeware/proposal/discussion/370-governance-working-group
Validators	commonwealth.im/edgeware-validators/
Builders	commonwealth.im/edgeware/proposal/discussion/371-builders-working-group
Brand	commonwealth.im/edgeware/proposal/discussion/372-brand-working-group
Artists	commonwealth.im/edgeware/discussions/wg-artists
DAO	commonwealth.im/edgeware/discussions/wg-DAOs
Economics	commonwealth.im/edgeware/discussions/wg-economics

Games commonwealth.im/edgeware/discussions/wg-games

Partnerships commonwealth.im/edgeware/discussions/wg-partnerships

Develop



Edgeware Builders Guild

Edgeware has [wg-builders](#) group where you can discuss about your project, aims of developers community on Edgeware, usability and feasibility with Edgeware community developers.

Projects

Major part of developers projects can be found at [Edgeware's Builders Guild repositories](#) and you are welcome to contribute to [awesome-edgeware](#) where we map Edgeware ecosystem projects

Engage

You can follow [Edgeware Builders Guild](#) at Twitter, join and engage with them at [Telegram](#), [Element](#), [Discord](#)

Learn

Edgeware has support for EVM Dapps and for ink! smart contract pallet as well. You can learn [how to play with EVM](#) or [learn how to deploy your first ink! smart contract](#).

If you are looking for advanced topics, you can play around with prepared examples, like `erc721`, `multisig_plain`, `delegator`, `dns`

Funding

If you have project in terms of User Interface, System Integrations, Tools, Research, Application-specific you can apply to [Construction Projects](#) for grant funding your project.

Resources

- [Join Edgeware community](#)
- [Substrate News and Resources](#)
- [Solang Solidity-Substrate Compiler](#)

Edgeware Core

Install Edgeware

To prepare setup to play around with Edgeware, you can learn more at [Edgeware node repository]((<https://github.com/hicommonwealth/edgeware-node/>))

Resources

- [Edgeware Builders Guild](#)
- [Node Repo](#)
- [Homepage Repo](#)
- [Docs Repo](#)

Edgeware Smart Contracts

Introduction to the Edgeware Contracts Workshop

Edgeware is blockchain platform with support [WASM Contracts Pallet](#) and [EVM Ethereum compatibility layer](#).

This is a self-guided tutorial which will teach you how to build both [WASM](#) and EVM-based smart contracts on [Edgeware](#).

In the WASM tutorials, we will cover:

1. Installing prerequisites on your computer
2. Using the ink! CLI to start a new project
3. Building and testing your contract
4. Deploying your contract on a local Edgeware node
5. Interacting with your contract using the Polkadot UI
6. Creating a basic flipper contract
7. Creating a simple incrementer contract
8. Creating a more complex ERC20 contract

In the [EVM tutorials](#), we will cover:

1. Setting up an Edgeware node
2. Setting up Metamask to work with Edgeware
3. Using Remix as an Ethereum IDE to compile, deploy, and call a contract
4. Using Truffle to deploy a contract to Edgeware
5. Using Web3.js to interact with Edgeware contracts programmatically

We would love for you to give feedback in these early stages, so please feel free to open a PR or comment on Commonwealth!

How to participate in the Edgeware Developers Guild - builders community

We have links to all resources accessible from one point: linktr.ee/edg_developers.

All our rooms are bridged so you can come in comfortable with your preferred chat platform.

Chats - Technical

- Telegram: EDG_Developers
- Element: Edgeware Builders

Chats - Governance

- Element: Edgeware Governacne
- Telegram: Edgeware GWG / Council
- Telegram: Edgeware General
- Telegram: Edgeware Validators

Social

- Twitter.com/heyedgeware
 - Twitter.com/edg_developers
-

Resources

- Smart Contracts vs Runtime Modules
- Parity Ink! Smart Contract Language
- Ink! Docs
- WebAssembly (Wasm)
- Substrate EVM Module
- Ink! Tutorial

WASM Setup

Introduction

This chapter will teach you everything you need to know to get started building smart contracts on Edgeware with ink!.

We will cover:

1. Installing prerequisites on your computer
2. Using the ink! `cargo-contract` plugin to start a new project
3. Building and testing our contract
4. Deploying our contract on a local Edgeware node
5. Interacting with our contract using the Polkadot UI

Setup Environment

To follow this tutorial, you will need to set up some stuff on your computer.

Substrate Prerequisites

To get started, you need to make sure your computer is set up to build Substrate. If you are using **OSX** or most popular **Linux** distros, you can do it by running:

```
curl https://sh.rustup.rs -ssf | sh -s -- -y
```

```
1 rustup target add wasm32-unknown-unknown --toolchain stable
2 rustup component add rust-src --toolchain nightly
3 rustup toolchain install nightly-2020-06-01
4 rustup target add wasm32-unknown-unknown --toolchain nightly-2020-06-01
```

The final tool we will be installing is the `ink!` command line utility which will make setting up Substrate smart contract projects easier.

You can install the utility using Cargo with:

```
cargo install --git https://github.com/hicommonwealth/cargo-contract cargo-co
```

You can then use `cargo contract --help` to start exploring the commands made available to you.



Note: The `ink!` CLI is under heavy development and some of its commands are not implemented, yet!

Creating an ink! Project

We are going to use the ink! CLI to generate the files we need for a Substrate smart contract project.

Make sure you are in your working directory, and then run:

```
cargo contract new flipper
```

This command will create a new project folder named `flipper` which we will explore:

```
cd flipper/
```

ink! Contract Project

```
1 flipper
2 |
3 .
4 └── Cargo.toml
5   └── lib.rs
```

Contract Source Code

The ink CLI automatically generates the source code for the "Flipper" contract, which is about the simplest "smart" contract you can build. You can take a sneak peak as to what will come by looking at the source code here:

[Flipper Example Source Code](#)

The Flipper contract is nothing more than a `bool` which gets flipped from true to false through the `flip()` function. We won't go so deep into the details of this source code because we will be walking you through the steps to build a more advanced contract!

Testing Your Contract

You will see at the bottom of the source code there is a simple test which verifies the functionality of the contract. We can quickly test that this code is functioning as expected using the **off-chain test environment** that ink! provides.

In your project folder run:

```
cargo +nightly test
```

To which you should see a successful test completion:

```
1 $ cargo +nightly test
2     running 2 tests
3     test flipper::tests::default_works ... ok
4     test flipper::tests::it_works ... ok
5
6     test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
```

Now that we are feeling confident things are working, we can actually compile this contract to Wasm.

Building Your Contract

Run the following command to compile your smart contract:

```
cargo +nightly contract build
```

This special command will turn your ink! project into a Wasm binary which you can deploy to your chain. If all goes well, you should see a `target` folder which contains this `.wasm` file.

```
1 target  
2 └── flipper.wasm
```

Contract Metadata

By running the next command we'll generate the contract metadata (a.k.a. the contract ABI):

```
cargo +nightly contract generate-metadata
```

You should have a new JSON file (`metadata.json`) in the same target directory:

```
1 target  
2 └── flipper.wasm  
3   └── metadata.json
```

Let's take a look at the structure inside:

```
1   {
2     "metadataVersion": "0.1.0",
3     "source": {
4       "hash": "0x11ba777b3457bf64cb99421786c90d421afde1c56579aa3dae336e58ccc8",
5       "language": "ink! 3.0.0-rc1",
6       "compiler": "rustc 1.47.0-nightly"
7     },
8     "contract": {
9       "name": "flipper",
10      "version": "0.1.0",
11      "authors": [
12        "[your_name]"
13      ]
14    },
15    "spec": {
16      "constructors": [
17        {
18          "args": [
19            {
20              "name": "init_value",
21              "type": {
22                "displayName": [
23                  "bool"
24                ],
25                "type": 1
26              }
27            }
28          ],
29          "docs": [
30            " Constructor that initializes the `bool` value to the given `init_value`."
31          ],
32          "name": [
33            "new"
34          ],
35          "selector": "0xd183512b"
36        },
37        {
38          "args": [],
39          "docs": [
40            " Constructor that initializes the `bool` value to `false`.",
41            "",
42            " Constructors can delegate to other constructors."
43          ],
44          "name": [
45            "default"
46          ],
47          "selector": "0x6a3712e2"
48        }
49      ],
50      "docs": []
51    }
52  }
53 }
```



```
103     "types": [
104         {
105             "def": {
106                 "primitive": "bool"
107             }
108         }
109     ]
110 }
```

You can see that this file describes all the interfaces that can be used to interact with your contract.

- Registry provides the **strings** and custom **types** used throughout the rest of the JSON.
- Storage defines all the **storage** items managed by your contract and how to ultimately access them.
- Contract stores information about the callable functions like **constructors** and **messages** a user can call to interact with your contract. It also has helpful information like the **events** that are emitted by the contract or any **docs**.

If you look close at the constructors and messages, you will also notice a `selector` which is a 4-byte hash of the function name and is used to route your contract calls to the correct functions.

Learn More

ink! provides a built-in overflow protection enabled on our `Cargo.toml` file. It is recommended to keep it enabled to prevent potential overflow errors in your contract.

```
1 [profile.release]
2 panic = "abort"           <-- Panics shall be treated as aborts: reduces b
3 lto = true                <-- enable link-time-optimization: more efficient
4 opt-level = "z"            <-- Optimize for small binary output
5 overflow-checks = true    <-- Arithmetic overflow protection
```

Running an Edgeware Node

We want to provide a fast setup experience for you. If you have Docker, you can launch an Edgeware development node in a few seconds:

Note If you don't have **Docker** installed, you can quickly install it from [here](#)

```
1 git clone https://github.com/hicommonwealth/edgeware-node; cd edgeware-node  
2 docker-compose up
```

i Note: If you have run this command in the past, you probably want to purge your chain storage, so that you run through this tutorial with a clean slate. You can do this easily by using `docker-compose rm` to delete your existing docker volume.

```
Creating docker_edgeware_1 ...  
Creating docker_edgeware_1 ... done  
Attaching to docker_edgeware_1  
edgeware_1 | 2021-02-16 17:23:22 It isn't safe to expose RPC publicly without a proxy server that filters available set of RPC methods.  
edgeware_1 | 2021-02-16 17:23:22 Edgeware Node  
edgeware_1 | 2021-02-16 17:23:22 * version 3.2.0-b9593b0-x86_64-linux-gnu  
edgeware_1 | 2021-02-16 17:23:22 * by Commonwealth Labs <hello@commonwealth.im>, 2017-2021  
edgeware_1 | 2021-02-16 17:23:22 └─ Chain specification: Development  
edgeware_1 | 2021-02-16 17:23:22 └─ Node name: exclusive-quiver-2501  
edgeware_1 | 2021-02-16 17:23:22 └─ Role: AUTHORITY  
edgeware_1 | 2021-02-16 17:23:22 └─ Database: RocksDb at /home/runner/.local/share/edgeware/chains/dev/db  
edgeware_1 | 2021-02-16 17:23:22 └─ Native runtime: edgeware-45 (edgeware-node-45-tx1.au16)  
edgeware_1 | 2021-02-16 17:23:22 └─ new validator set of size 1 has been elected via ElectionCompute::OnChain for era 0  
edgeware_1 | 2021-02-16 17:23:22 └─ Initializing Genesis block/state (state: 0x4184..0aae, header-hash: 0x7f36..399a)  
edgeware_1 | 2021-02-16 17:23:22 └─ Loading GRANDPA authority set from genesis on what appears to be first startup.  
edgeware_1 | 2021-02-16 17:23:22 └─ Loaded block-time = 6000 milliseconds from genesis on first-launch  
edgeware_1 | 2021-02-16 17:23:22 └─ Using default protocol ID "sup" because none is configured in the chain specs  
edgeware_1 | 2021-02-16 17:23:22 └─ Local node identity is: 12D3KooWLGwng87ishah9vdgvyDcZzLXh3gWHiugRXScrLXh3gK1Y  
edgeware_1 | 2021-02-16 17:23:22 └─ Highest known block at #0  
edgeware_1 | 2021-02-16 17:23:22 └─ Prometheus server started at 127.0.0.1:9615  
edgeware_1 | 2021-02-16 17:23:22 └─ Listening for new connections on 0.0.0.0:9944.  
edgeware_1 | 2021-02-16 17:23:24 └─ Starting consensus session on top of parent 0x7f36e96bleb14af0fb9f09f42111e4d5501f559912dfe0b778fdbd3ee3c39  
9a  
edgeware_1 | 2021-02-16 17:23:24 └─ Prepared block for proposing at 1 [hash: 0x176c371457f47daf2a64ba65376c4318774d6020ef3663bffaeb8221cf56bd41;  
parent_hash: 0x7f36..399a]; extrinsics (1): [0x7711..3ce3]  
edgeware_1 | 2021-02-16 17:23:24 └─ Pre-sealed block for proposal at 1. Hash now 0x09c8a078a19faae7ad02121a7cbd767c25433df3ca6c172a8d993cd2b7d17  
756, previously 0x176c371457f47daf2a64ba65376c4318774d6020ef3663bffaeb8221cf56bd41.  
edgeware_1 | 2021-02-16 17:23:24 └─ Imported #1 (0x09c8..7756)  
edgeware_1 | 2021-02-16 17:23:27 └─ Idle (0 peers), best: #1 (0x09c8..7756), finalized #0 (0x7f36..399a), + 0 : 0  
edgeware_1 | 2021-02-16 17:23:30 └─ Starting consensus session on top of parent 0x09c8a078a19faae7ad02121a7cbd767c25433df3ca6c172a8d993cd2b7d177  
56  
edgeware_1 | 2021-02-16 17:23:30 └─ Prepared block for proposing at 2 [hash: 0x0c60b07b870a94d5e33b8973e95ee83b27fcabf1478ff1b76deeb54506d401a;  
parent_hash: 0x09c8..7756; extrinsics (1): [0x47b7..d85d]]  
edgeware_1 | 2021-02-16 17:23:30 └─ Pre-sealed block for proposal at 2. Hash now 0xf4f5412d3235ee40a979987542dbace26657920f0cda2ee1d8fd1f633f730  
1fa, previously 0x0c60b07b870a94d5e33b8973e95ee83b27fcabf1478ff1b76deeb54506d401a.  
edgeware_1 | 2021-02-16 17:23:30 └─ Imported #2 (0xf4f5..01fa)  
edgeware_1 | 2021-02-16 17:23:32 └─ Idle (0 peers), best: #2 (0xf4f5..01fa), finalized #0 (0x7f36..399a), + 0 : 0  
edgeware_1 | 2021-02-16 17:23:36 └─ Starting consensus session on top of parent 0xf4f5412d3235ee40a979987542dbace26657920f0cda2ee1d8fd1f633f7301  
fa  
edgeware_1 | 2021-02-16 17:23:36 └─ Prepared block for proposing at 3 [hash: 0xb12ac587b8e6e9e1ef67848c980254c95c3e8ac2c3de88a14f7b528943ef6bc3;  
parent_hash: 0xf4f5..01fa; extrinsics (1): [0x09e2..24a1]]  
edgeware_1 | 2021-02-16 17:23:36 └─ Pre-sealed block for proposal at 3. Hash now 0xb2f58ba1501daf54a973e79edb4162c456c0c9a8131e62b243e105dde598e  
e31, previously 0xb12ac587b8e6e9e1ef67848c980254c95c3e8ac2c3de88a14f7b528943ef6bc3.  
edgeware_1 | 2021-02-16 17:23:36 └─ Imported #3 (0xb2f5..ee31)  
edgeware_1 | 2021-02-16 17:23:37 └─ Idle (0 peers), best: #3 (0xb2f5..ee31), finalized #1 (0x09c8..7756), + 0 : 0
```

You should start to see blocks being produced by your node in your terminal.

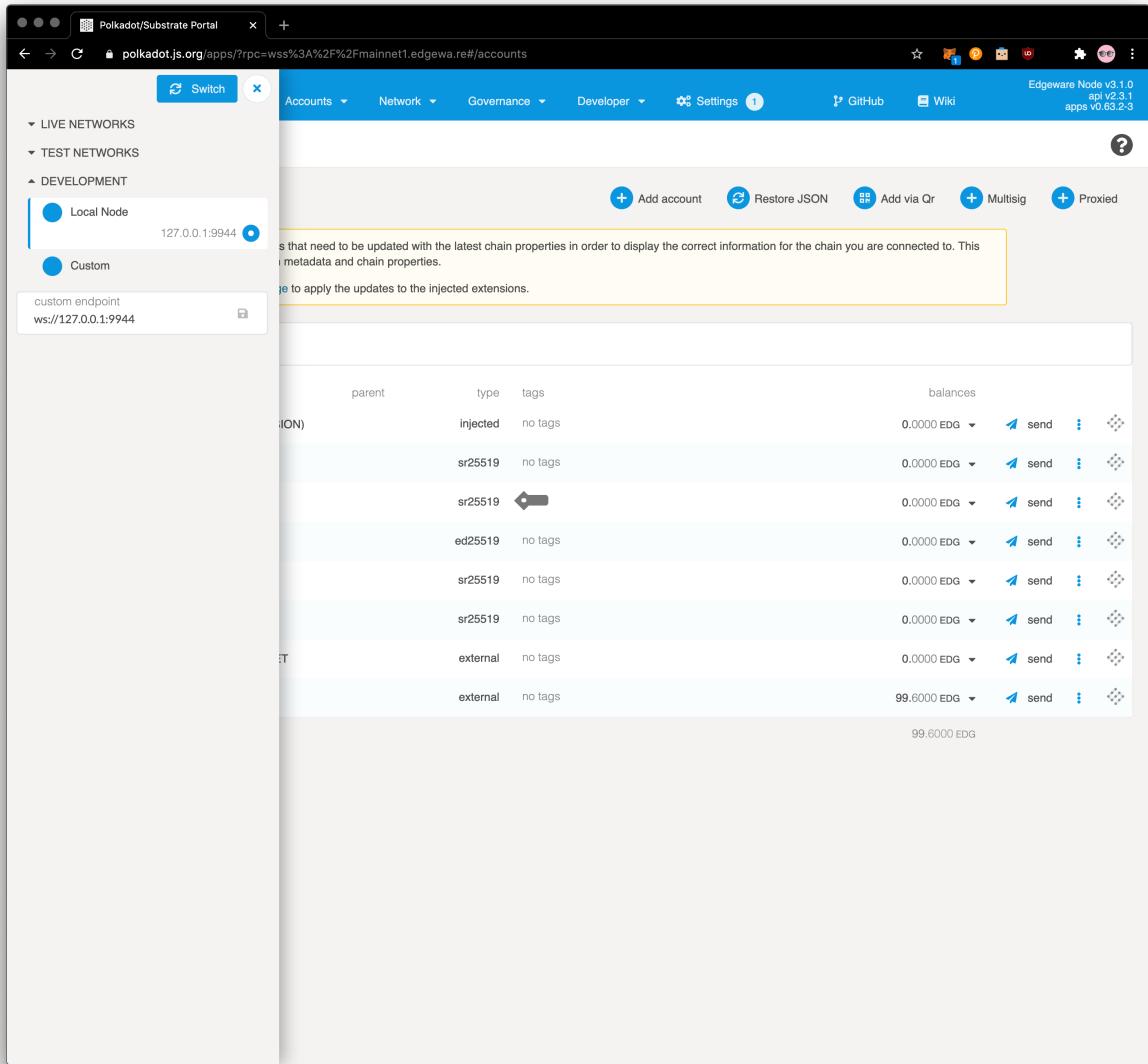
You can interact with your node using the Polkadot UI:



Note: You will need to use a Chromium based browser (Google Chrome) to have this site interact with your local node. The Polkadot UI is hosted on a secure server, and your local node is not, which may cause compatibility issues on Firefox. The other option is to [clone and run the Polkadot UI locally](#). (The Brave Browser has also been causing compatibility issues).

To point the UI to your local node, you need to adjust the **Settings**. Just select 'Local Node (127.0.0.1:9944)' from the endpoint dropdown:

Click on the chain name > Development node/endpoint to connect to > Local Nod



If you go into the **Explorer** tab of the UI, you should also see blocks being produced!

Polkadot/Substrate Portal

polkadot.js.org/apps/?rpc=ws%3A%2F%2F127.0.0.1%3A9944#/explorer

Development version 41 #11 Explorer Accounts Network Governance Developer Settings GitHub Wiki

Edgeware Node v3.1.0 api v2.3.1 apps v0.63.2-3

Chain info Block details Node info block hash or number to query

last block 4.2 s target 6 s total issuance 1.4000 GtEDG session #0 finalized 9 best 11

recent blocks

11	0xfe77eaf5127fb47036a99f2e559f32741ef2713a...	ALICE_STASH
10	0x8d979ac0b486edc2b9d85bec6f2369928c926e34...	ALICE_STASH

recent events

treasuryReward.TreasuryMinting	11-2
treasuryReward.TreasuryMinting	10-2

The screenshot shows the Polkadot/Substrate Portal interface. At the top, there's a navigation bar with tabs for Development (version 41 #11), Explorer, Accounts, Network, Governance, Developer, Settings, GitHub, and Wiki. The Explorer tab is selected. On the right side of the header, it says "Edgeware Node v3.1.0 api v2.3.1 apps v0.63.2-3". Below the header, there are three main sections: "Chain info", "Block details", and "Node info". The "Chain info" section displays metrics like last block (4.2 s), target (6 s), total issuance (1.4000 GtEDG), session (#0), finalized (9), and best (11). Below these are two tables under "recent blocks" and "recent events". The "recent blocks" table has two entries: block 11 with hash 0xfe77eaf5127fb47036a99f2e559f32741ef2713a... and block 10 with hash 0x8d979ac0b486edc2b9d85bec6f2369928c926e34... Both entries are associated with the "ALICE_STASH" account. The "recent events" table also has two entries: treasuryReward.TreasuryMinting at height 11-2 and treasuryReward.TreasuryMinting at height 10-2.

Deploying Your Contract

Now that we have generated the Wasm binary from our source code and started a Substrate node, we want to deploy this contract onto our Substrate blockchain.

Smart contract deployment on Substrate is a little different than on traditional smart contract blockchains.

Whereas a completely new blob of smart contract source code is deployed each time you push a contract on other platforms, Substrate opts to optimize this behavior. For example, the standard ERC20 token has been deployed to Ethereum thousands of times, sometimes only with changes to the initial configuration (through the Solidity `constructor` function). Each of these instances take up space on the blockchain equivalent to the contract source code size, even though no code was actually changed.

In Substrate, the contract deployment process is split into two halves:

1. Putting your code on the blockchain
2. Creating an instance of your contract

With this pattern, contract code like the ERC20 standard can be put on the blockchain a single time, but instantiated any number of times. No need to continually upload the same source code over and waste space on the blockchain.

Fast Track

If you skipped previous steps and just want to see interaction with ink! smart contract, download [flipper.wasm](#) and [metadata.json](#)

Putting Your Code on the Blockchain

With your Substrate development node running, you can go back to the [Polkadot UI](#) where you will be able to interact with your blockchain.

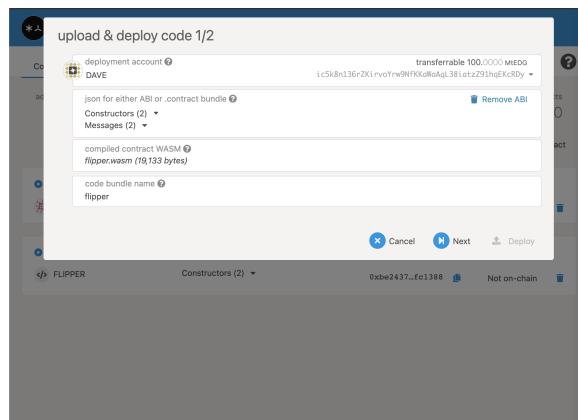
Under the **Developer** tab click on the specially designed **Contracts** section of the UI.

In the **Contracts** section, select "**Upload & deploy code**".

The screenshot shows the Tezos UI interface. At the top, there is a navigation bar with tabs: Accounts, Network, Governance, Developer (which is currently selected), and Settings. Below the navigation bar, the main area is titled "Contracts". It displays two sections: "addresses" (0) and "code hashes" (0). In the center, there are three buttons: "+ Upload & deploy code" (highlighted with a black arrow pointing to it), "+ Add an existing code hash", and "+ Add an existing contract". Below these buttons, there are two expandable sections: "contracts" (underlined) and "code hashes". Each section contains a table with columns: name, status, and a trash icon. The "contracts" section shows one entry: "FLIPPER" (status: Not on-chain). The "code hashes" section shows one entry: "FLIPPER" (status: Not on-chain).

In the popup, select a **deployment account** with some account balance, like `Dave`. For the **contract's metadata**, select the JSON file. In **compiled contract WASM**, select the `flipper.wasm` file we generated. The code bundle name will be "flipper". Once all the

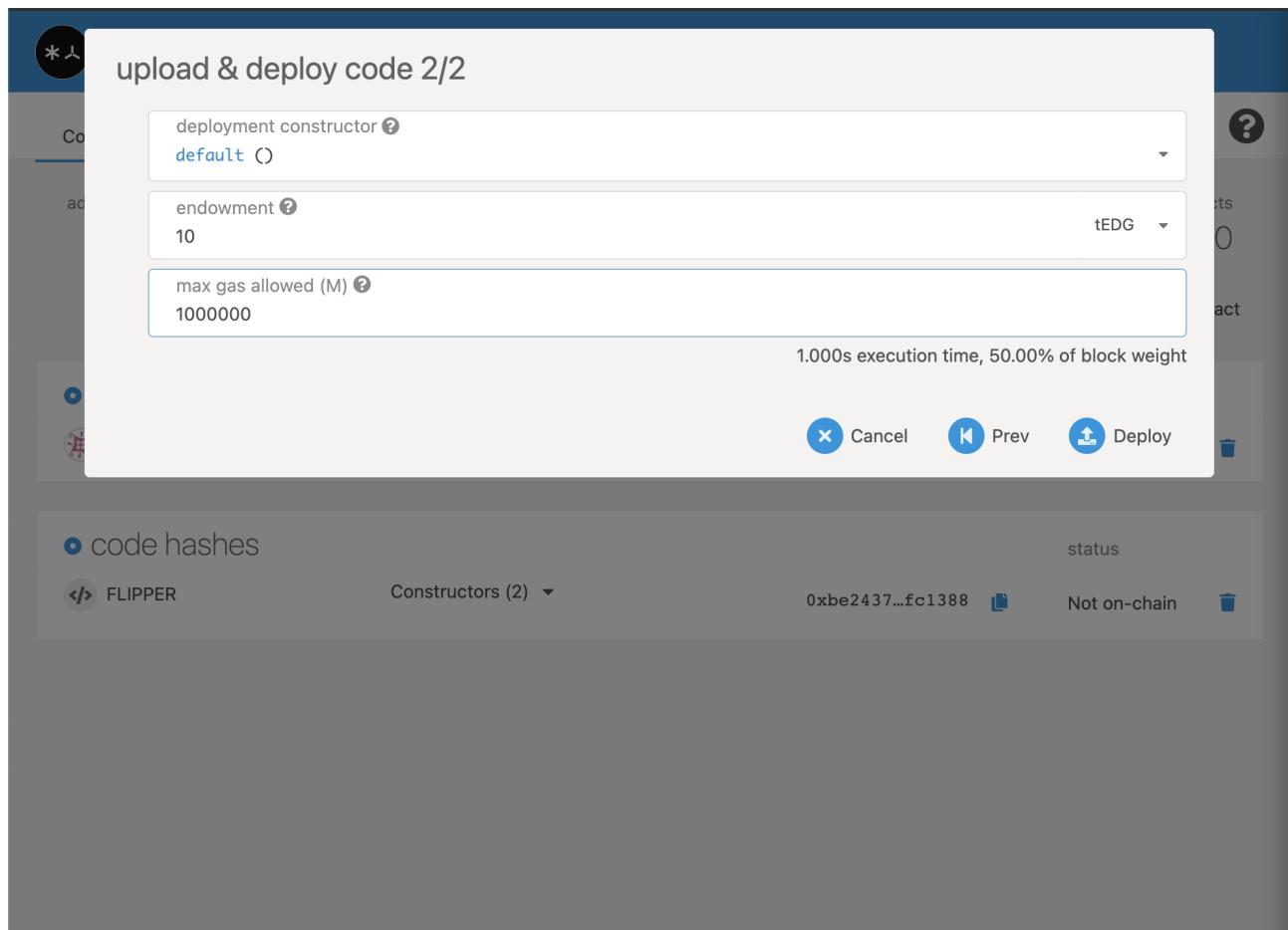
parameters are filled, click **Next**.



Creating an Instance of Your Contract

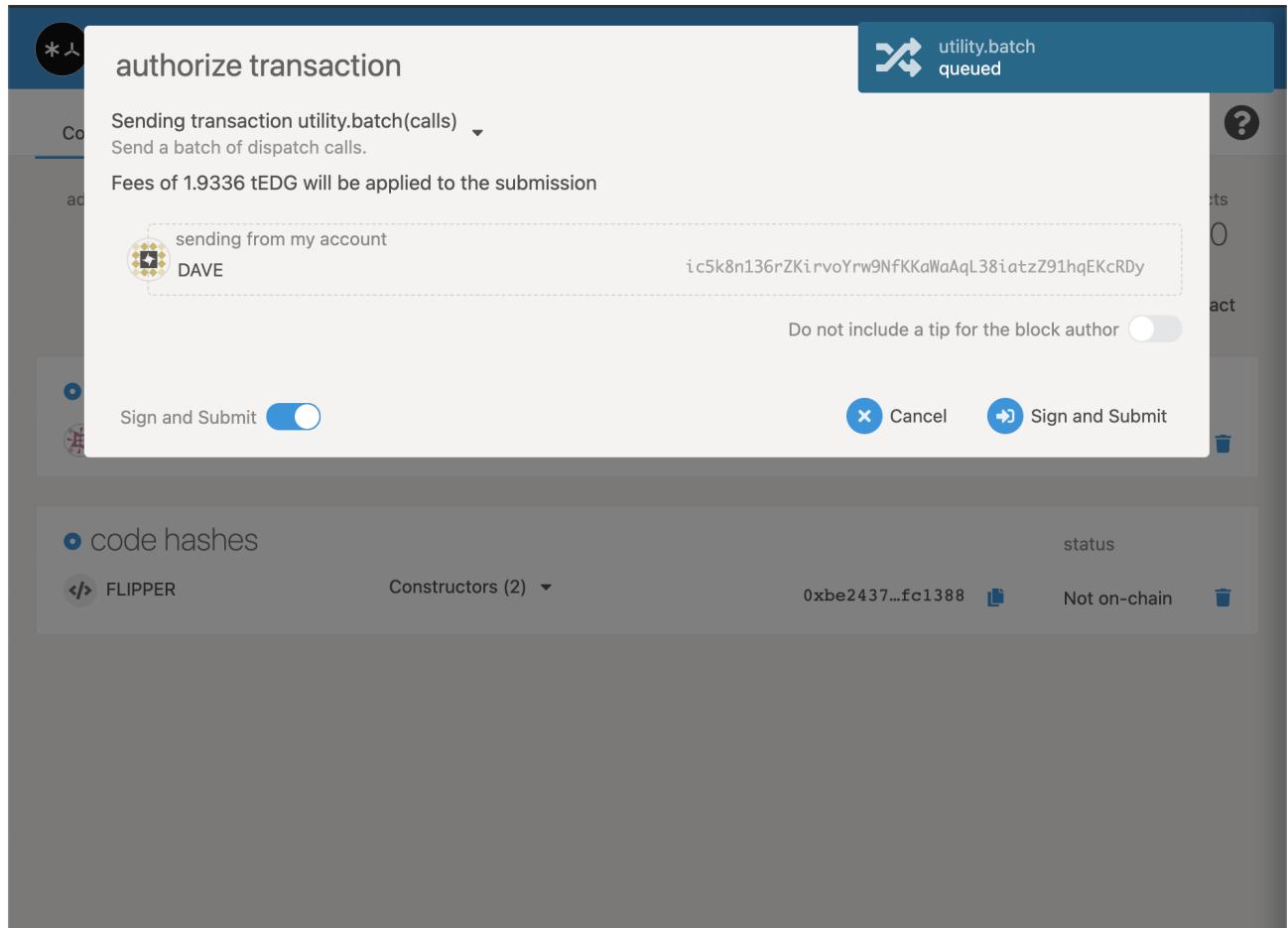
Smart contracts exist as an extension of the account system on the blockchain. Thus creating an instance of this contract will create a new `AccountId` which will store any balance managed by the smart contract and allow us to interact with the contract.

To instantiate our contract we just need to give this contract account an ***endowment*** of **10 Units** in order to pay the storage rent and set the ***maximum gas allowed*** to value(**1,000,000**):



i Note: As mentioned earlier, contract creation involves creation of a new Account. As such, you must be sure to give the contract account at least the existential deposit defined by your blockchain. We also need to be able to pay the contract's rent (`endowment`). If we consume all of this deposit, the contract will become invalid. We can always refill the contract's balance and keep it on chain.

You will then **authorize** the contract, by **signing and submitting**. You can also choose to leave a tip for the block author if you'd like.



When you press **Deploy**, you should see a flurry of events appear including the creation of a new account (`system.NewAccount`) and the instantiation of the contract (`contracts.instantiate`):

The screenshot shows the Substrate Node UI interface. At the top, there is a navigation bar with tabs for Contracts, Accounts, Network, Governance, and Dev. A notification bar at the top right displays a green message: "utility.batch inblock" with a checkmark icon and a close button (X). Below the navigation bar, the main content area is divided into sections for Contracts and code hashes.

Contracts

addresses: 1

Upload & deploy code | Add an existing

contracts

		status
	FLIPPER	Messages (2) ▾ 0.0000 tEDG ▾ Not on-chain [copy] [trash]
	FLIPPER	Messages (2) ▾ 10.0000 tEDG ▾ Alive [copy] [trash]

code hashes

		status
	FLIPPER	Constructors (2) ▾ 0xbe2437...fc1388 [copy] Available [copy] [trash]

Calling Your Contract

Now that your contract has been fully deployed, we can start to interact with it! Flipper only has two functions, so we will show you what it's like to play with both of them.

get()

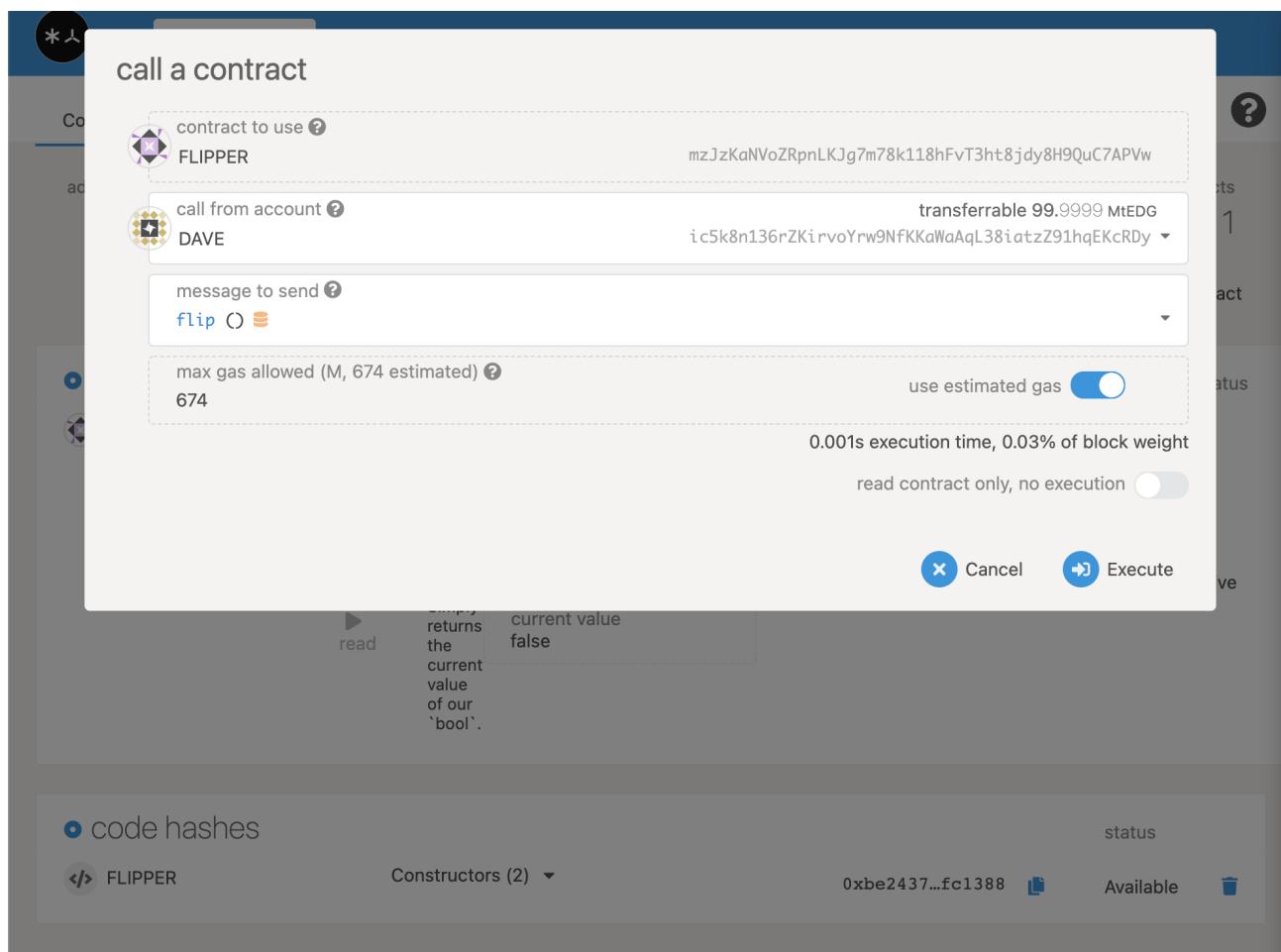
If you take a look back at our contract's `on_deploy()` function, we set the initial value of the Flipper contract to `false`. Let's check that this is the case.

The screenshot shows the Flipper application interface. At the top, there is a navigation bar with tabs for Contracts, Accounts, Network, Governance, Developer, and Settings. The Contracts tab is selected. Below the navigation bar, there are sections for addresses, code hashes, and contracts, each showing a count of 1. There are three buttons: 'Upload & deploy code', 'Add an existing code hash', and 'Add an existing contract'. The main area shows a list of messages for the 'FLIPPER' contract. The 'exec flip' message is described as a message that can be called on instantiated contracts. The 'read get' message is described as simply returning the current value of the 'bool'. A tooltip for the 'get' message shows the current value as 'false'. In the bottom section, under 'code hashes', the 'FLIPPER' contract is listed with a status of 'Available'.

flip()

So let's make the value `true` now!

The alternative **message to send** we can make with the UI is `flip()`. In this updated version, the contract will automatically estimate the necessary **max gas allowed**. In this case, it's **674**. Leave estimated gas turned on, and **execute** the contract. In the following screen, authorize the transaction by clicking the `sign` and `submit` button.



You will notice that this call actually sends a transaction. If the transaction was successful, we should then be able to go back to the `get()` function and see our updated storage:

The screenshot shows the Edge Network Dev Portal interface. At the top, there's a navigation bar with tabs for Contracts, Accounts, Network, Governance, Developer, and Settings. Below the navigation bar, the main area is titled "Contracts". It displays a summary: 1 address, 1 code hash, and 1 contract. There are three buttons for actions: "Upload & deploy code", "Add an existing code hash", and "Add an existing contract".

The contracts section lists one entry: "FLIPPER". It has 2 messages: "exec" and "get". The "exec" message is described as "A message that can be called on instantiated contracts." The "get" message is described as "Simply returns the current value of our 'bool'." A callout box highlights the "get" message with the text "current value true".

The code hashes section lists one entry: "FLIPPER". It has 2 constructors: "Constructors (2)". The status is "Available".

Woohoo! You deployed your first smart contract!

Moving Forward

We will not go over these setup and deployment steps again, but we will use them throughout the tutorial. You can always come back to this chapter if you need to remember how to do a certain process.

The rest of the tutorial will have **template code** which you will use to walk through the different steps of contract development. Each template comes with a fully designed suite of tests that should pass if you programmed your contract correctly. Before you move on from a section, make sure that you run:

```
cargo +nightly test
```

and that the tests all execute successfully, without any warnings.

You need not deploy your contract between each section, but if we ask you to deploy your contract, you will need to follow the same steps you have done with the Flipper contract.

Live Smart Contracts

Live Smart Contracts - Beresheet

Edgeware has magical functionality, that you will **upload your contract once** and you can **instantiate many times**.

-  **Caching.** Contracts are cached by default and therefore means they only need to be deployed once, and afterward be instantiated as many times as you want. This helps to keep the storage load on the chain down to the minimum. On top of this, when a contract is no longer being used and the existential deposit is drained, the code will be erased from storage (known as reaping).

We have deployed few contracts for you to the [Beresheet - our testnet network \(open Apps\)](#) to make it easy play for you. You will just need download particular metadata (ABI) for particular contract so your browser understands it.

-  [You can find metadata here](#)
-  [Faucet](#)
-  [If faucet doesn't work, visit us at Telegram](#)

Code hashes

- erc20

0x3d9c89932759e97326550ae1ede0efc9ff570e170645ba0538da79a3129a6720

- erc721

0xa30ed09d749f8647f63dbc40fad83b278cdc559968b4aece805f6514ccce357e

- incrementer

0xa0de565eb4be2fdaac56cb1dc0fd6199d0f1f3c5cc1d53929b6f47ea4b6c9afb

- multisig_plain

0xe93c49ad335a33202bd5e4c6d250495d12f424e655da330a2fd90c97c1cd7938

- trait-erc20

0x1ab1a476dc17e3ba900cc66098998a659a345ce5075e6b7f97cb6e0000a9ab4d

- trait-flipper

0x93ec7d302ff5ecc5ca8f0602805d6068f036bfaa2bd7883e2f550a919b8ee735

- dns

0xa0c4b6d7f5428cb74e4dc771dbdf8f2ee588cb58d14a24f99a798fc82d6687cb

- delegator

0x08e221640f02aba842be94139dc4f1c5048605f25ec8d85a13b60bd4e15779ea

- contract terminate

```
0xd67d39fe76eae342b621bd79f3b70c51f2c744c1c5f3f01b66609f412be7bc8f
```

Add Code hashes

[Navigate to the contracts](#)



Copy one of code hashes from above, put them in code hash, name it and choose metadata, that json one, it should look like this



Reach us for more engagement

Glad you've made it through! ☺ We are eager to guide your more on your exploration through Edgeware ink! combability feature. We are **keen to hear your experience and suggestion you may have for us..** You can feel free to [chat with us in the Edgeware's channels like Discord, Element and Telegram](#), we can help you out with issues you may have or project you may want to be funded through our [Treasury program](#). Don't hesitate to share your feedback on our channels, there is always space to improve! ☺

WASM Basics

Introduction

This chapter will get you started developing smart contracts with ink!.

We will build a simple "Incrementer" contract which holds a number which you can increase with a function call.

Over the course of this chapter you will learn:

- The structure of ink! smart contracts
- To store single values and hash maps
- To safely get and set these values
- To build public and private functions
- To configure Rust to use safe math

Contract Template

Let's take a look at a high level what is available to you when developing a smart contract using the ink!.

ink!

ink! is an [eDSL](#) to write WebAssembly based smart contracts in the Rust programming language.

ink! is just standard Rust in a well defined "contract format" with specialized `#[ink(...)]` attribute macros. These attribute macros tell ink! what the different parts of your Rust smart contract represent, and ultimately allows ink! to do all the magic needed to create Substrate compatible Wasm bytecodes!

Your Turn!

We are going to start a new project for the Incrementer contract we will build in this chapter.

So go into your working directory and run:

```
cargo contract new incrementer
```

Just like before, this will create a new project folder named `incrementer` which we will use for the rest of this chapter.

```
cd incrementer/
```

In the `lib.rs` file, replace the "Flipper" contract source code with the template code provided here.

Quickly check that it compiles and the trivial test passes with:

```
cargo +nightly test
```

Also check that you can build the Wasm file by running:

```
cargo +nightly contract build
```

If everything looks good, then we are ready to start programming!

✓Potential Solution

```
lib.rs

1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod incrementer {
7
8      #[ink(storage)]
9      pub struct Incrementer {
10          // Storage Declaration
11      }
12
13      impl Incrementer {
14          #[ink(constructor)]
15          pub fn new(init_value: i32) -> Self {
16              // Contract Constructor
17              Self{}
18          }
19
20          #[ink(message)]
21          pub fn get(&self) {
22              // Contract Message
23          }
24      }
25
26      #[cfg(test)]
27      mod tests {
28          #[test]
29          fn default_works() {
30              // Test Your Contract
31          }
32      }
33 }
```

Storing a Value

The first thing we are going to do to the contract template is introduce some storage values.

Here is how you would store some simple values in storage:

```
1 #[ink(storage)]
2 pub struct MyContract {
3     // Store a bool
4     my_bool: bool,
5     // Store some number
6     my_number: u32,
7 }
8 /* --snip-- */
```

Supported Types

Contract may store types that are encodable and decodable with [Parity Codec](#) which includes the most common types such as `bool` , `u{8,16,32,64,128}` , `i{8,16,32,64,128}` , `String` , tuples, and arrays.

ink! provides smart contracts Substrate specific types like `AccountId` , `Balance` , and `Hash` as if they were primitive types. Also ink! provides storage types for more elaborate storage interactions through the storage module:

```
use ink_storage::collections::{Vec, HashMap, Stash, Bitvec};
```

Here is an example of how you would store an `AccountId` and `Balance` :

```
1 // We are importing the default ink! types
2 use ink_lang as ink;
3
```

```
4 #[ink::contract]
5 mod MyContract {
6
7     // Our struct will use those default ink! types
8     #[ink(storage)]
9     pub struct MyContract {
10         // Store some AccountId
11         my_account: AccountId,
12         // Store some Balance
13         my_balance: Balance,
14     }
15     /* --snip-- */
16 }
```

You can find all the supported Substrate types in [crates/storage/src/lib.rs](#).

Contract Deployment

Every ink! smart contract must have a constructor which is run once when a contract is created. ink! smart contracts can have multiple constructors:

```
1 use ink_lang as ink;
2
3 #[ink::contract]
4 mod mycontract {
5
6     #[ink(storage)]
7     pub struct MyContract {
8         number: u32,
9     }
10
11     impl MyContract {
12         /// Constructor that initializes the `u32` value to the given `init`.
13         #[ink(constructor)]
14         pub fn new(init_value: u32) -> Self {
15             Self {
16                 number: init_value,
17             }
18         }
19
20         /// Constructor that initializes the `u32` value to the `u32` default.
21         ///
22     }
23 }
```

```
22     /// Constructors can delegate to other constructors.
23     #[ink(constructor)]
24     pub fn default() -> Self {
25         Self {
26             number: Default::default(),
27         }
28     }
29     /* --snip-- */
30 }
31 }
```

Your Turn!

Follow the `ACTION`s in the template.

Remember to run `cargo +nightly test` to test your work.

Starting Point

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod incrementer {
7
8      #[ink(storage)]
9      pub struct Incrementer {
10          // ACTION: Create a storage value called `value` which holds .
11      }
12
13     impl Incrementer {
14         #[ink(constructor)]
15         pub fn new(init_value: i32) -> Self {
16             // ACTION: Create a new `Incrementer` and set its `value` .
17         }
18
19         // ACTION: Create a second constructor function named `default`.
20         //           It has no input, and creates a new `Incrementer` w
21         //           set to `0`.
22
23         #[ink(message)]
24         pub fn get(&self) {
25             // Contract Message
26         }
27     }
28
29     #[cfg(test)]
30     mod tests {
31         use super::*;

32         #[test]
33         fn default_works() {
34             Incrementer::default();
35         }
36     }
37 }
38 }
```

✓Potential Solution

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod incrementer {
7      #[ink(storage)]
8      pub struct Incrementer {
9          value: i32,
10     }
11
12     impl Incrementer {
13         #[ink(constructor)]
14         pub fn new(init_value: i32) -> Self {
15             Self {
16                 value: init_value,
17             }
18         }
19
20         #[ink(constructor)]
21         pub fn default() -> Self {
22             Self {
23                 value: 0,
24             }
25         }
26
27         #[ink(message)]
28         pub fn get(&self) {
29             // Contract Message
30         }
31     }
32
33     #[cfg(test)]
34     mod tests {
35         use super::*;

36
37         #[test]
38         fn default_works() {
39             Incrementer::default();
40         }
41     }
42 }
```

Getting a Value

Now that we have created and initialized a storage value, we are going to start to interact with it!

Contract Functions

As you can see in the contract template, all of your contract functions are part of your contract module.

```
1 impl MyContract {  
2     // Public and Private functions can go here  
3 }
```

Public and Private Functions

In Rust, you can make as many implementations as you want. As a stylistic choice, we recommend breaking up your implementation definitions for your private and public functions:

```
1 impl MyContract {  
2     /// Public function  
3     #[ink(message)]  
4     pub fn my_public_function(&self) {  
5         /* --snip-- */  
6     }  
7  
8     /// Private function  
9     fn my_private_function(&self) {  
10        /* --snip-- */  
11    }  
12  
13    /* --snip-- */  
14 }
```

You can also choose to split things up however is most clear for your project.

Note that all public functions must use the `#[ink(message)]` attribute.

Storage Value API

Without going into so much detail, storage values are a part of the underlying ink! core layer. In the background, they use a more primitive `cell` type which holds an `Option<T>`. When we try to get the value from storage, we `unwrap` the value, which is why it panics if it is not initialized!

```
1  impl<T> Value<T>
2  where
3      T: scale::Codec,
4  {
5      /// Returns an immutable reference to the wrapped value.
6      pub fn get(&self) -> &T {
7          self.cell.get().unwrap()
8      }
9
10     /// Returns a mutable reference to the wrapped value.
11     pub fn get_mut(&mut self) -> &mut T {
12         self.cell.get_mut().unwrap()
13     }
14
15     /// Sets the wrapped value to the given value.
16     pub fn set(&mut self, val: T) {
17         self.cell.set(val);
18     }
19 }
```

In that same file, you can find the other APIs exposed by storage values, however these three are the most commonly used.

Getting a Value

We already showed you how to initialize a storage value. Getting the value is just as simple:

```
1 impl MyContract {  
2     #[ink(message)]  
3     fn my_getter(&self) -> u32 {  
4         self.number  
5     }  
6 }
```

In Rust, if the last expression in a function does not have a semicolon, then it will be the return value.

Your Turn!

Follow the `ACTION`s on the code template provided.

Remember to run `cargo +nightly test` to test your work.

Starting Point

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod incrementer {
7
8      #[ink(storage)]
9      pub struct Incrementer {
10          value: i32,
11      }
12
13     impl Incrementer {
14         #[ink(constructor)]
15         pub fn new(init_value: i32) -> Self {
16             Self {
17                 value: init_value
18             }
19         }
20
21         #[ink(constructor)]
22         pub fn default() -> Self {
23             Self {
24                 value: 0
25             }
26         }
27
28         #[ink(message)]
29         // ACTION: Update this function to return an i32.
30         pub fn get(&self) {
31             // ACTION: Return the `value`.
32         }
33     }
34
35     #[cfg(test)]
36     mod tests {
37         use super::*;

38         #[test]
39         fn default_works() {
40             let contract = Incrementer::default();
41             assert_eq!(contract.get(), 0);
42         }
43     }
44 }
45 }
```


✓Potential Solution

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod incrementer {
7
8      #[ink(storage)]
9      pub struct Incrementer {
10          value: i32,
11      }
12
13     impl Incrementer {
14         #[ink(constructor)]
15         pub fn new(init_value: i32) -> Self {
16             Self {
17                 value: init_value
18             }
19         }
20
21         #[ink(constructor)]
22         pub fn default() -> Self {
23             Self {
24                 value: 0
25             }
26         }
27
28         #[ink(message)]
29         pub fn get(&self) -> i32 {
30             self.value
31         }
32     }
33
34     #[cfg(test)]
35     mod tests {
36         use super::*;

37         #[test]
38         fn default_works() {
39             let contract = Incrementer::default();
40             assert_eq!(contract.get(), 0);
41         }
42     }
43 }
44 }
```

Incrementing the Value

It's time to let our users modify storage!

Mutable and Immutable Functions

You may have noticed that the function templates included `self` as the first parameter of the contract functions. It is through `self` that you gain access to all your contract functions and storage items.

If you are simply *reading* from the contract storage, you only need to pass `&self`. But if you want to *modify* storage items, you will need to explicitly mark it as mutable, `&mut self`.

```
1  impl MyContract {
2      #[ink(message)]
3      pub fn my_getter(&self) -> u32 {
4          self.my_number
5      }
6
7      #[ink(message)]
8      pub fn my_setter(&mut self, new_value: u32) {
9          self.my_number = new_value;
10     }
11 }
```

Lazy Storage Values

There is a `Lazy` type that can be used for ink! storage values that don't need to be loaded in some or most cases. Because they do not meet this criteria, many simple ink! examples, including those in this workshop, do not require the use `Lazy` values. Since there is some overhead associated with `Lazy` values, they should only be used where required.

```
1 #[ink(storage)]
2 pub struct MyContract {
3     // Store some number
4     my_number: ink_storage::Lazy<u32>,
5 }
6
7 impl MyContract {
8     #[ink(constructor)]
9     pub fn new(init_value: i32) -> Self {
10         Self {
11             my_number: Default::default(),
12         }
13     }
14
15     #[ink(message)]
16     pub fn my_setter(&mut self, new_value: u32) {
17         ink_storage::Lazy::<u32>::set(&mut self.my_number, new_value);
18     }
19
20     #[ink(message)]
21     pub fn my_adder(&mut self, add_value: u32) {
22         let my_number = &mut self.my_number;
23         let cur = ink_storage::Lazy::<u32>::get(my_number);
24         ink_storage::Lazy::<u32>::set(my_number, cur + add_value);
25     }
26 }
```

Your Turn

Follow the `ACTION`s in the template code.

Remember to run `cargo +nightly test` to test your work.

□ Starting Point

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod incrementer {
7
8      #[ink(storage)]
9      pub struct Incrementer {
10          value: i32,
11      }
12
13     impl Incrementer {
14         #[ink(constructor)]
15         pub fn new(init_value: i32) -> Self {
16             Self {
17                 value: init_value,
18             }
19         }
20
21         #[ink(constructor)]
22         pub fn default() -> Self {
23             Self {
24                 value: 0,
25             }
26         }
27
28         #[ink(message)]
29         pub fn get(&self) -> i32 {
30             self.value
31         }
32
33         #[ink(message)]
34         pub fn inc(&mut self, by: i32) {
35             // ACTION: Simply increment `value` by `by`
36         }
37     }
38
39     #[cfg(test)]
40     mod tests {
41         use super::*;

42         #[test]
43         fn default_works() {
```

✓Potential Solution

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod incrementer {
7
8      #[ink(storage)]
9      pub struct Incrementer {
10          value: i32,
11      }
12
13     impl Incrementer {
14         #[ink(constructor)]
15         pub fn new(init_value: i32) -> Self {
16             Self {
17                 value: init_value,
18             }
19         }
20
21         #[ink(constructor)]
22         pub fn default() -> Self {
23             Self {
24                 value: 0,
25             }
26         }
27
28         #[ink(message)]
29         pub fn get(&self) -> i32 {
30             self.value
31         }
32
33         #[ink(message)]
34         pub fn inc(&mut self, by: i32) {
35             self.value += by;
36         }
37     }
38
39     #[cfg(test)]
40     mod tests {
41         use super::*;

42         #[test]
43         fn default_works() {
```

Storing a Mapping

Storing a Mapping

Let's now extend our Incrementer to not only manage one number, but to manage one number per user!

Storage HashMap

In addition to storing individual values, ink! also supports a `HashMap` which allows you to store items in a key-value mapping.

Here is an example of a mapping from user to a number:

```
1 #[ink(storage)]
2 pub struct MyContract {
3     // Store a mapping from AccountIds to a u32
4     my_number_map: ink_storage::collections::HashMap<AccountId, u32>,
5 }
```

This means that for a given key, you can store a unique instance of a value type. In this case, each "user" gets their own number, and we can build logic so that only they can modify their own number.

Storage HashMap API

You can find the full HashMap API in the [crates/storage/src/collections/hashmap](#) part of ink!.

Here are some of the most common functions you might use:

```
1     /// Inserts a key-value pair into the map.
2     ///
3     /// Returns the previous value associated with the same key if any.
4     /// If the map did not have this key present, `None` is returned.
```

```

5     pub fn insert(&mut self, key: K, new_value: V) -> Option<V> {/* --snip-- */
6
7     /// Removes the key/value pair from the map associated with the given key.
8     ///
9     /// - Returns the removed value if any.
10    pub fn take<Q>(&mut self, key: &Q) -> Option<V> {/* --snip-- */}
11
12   /// Returns a shared reference to the value corresponding to the key.
13   ///
14   /// The key may be any borrowed form of the map's key type,
15   /// but `Hash` and `Eq` on the borrowed form must match those for the key.
16   pub fn get<Q>(&self, key: &Q) -> Option<&V> {/* --snip-- */}
17
18   /// Returns a mutable reference to the value corresponding to the key.
19   ///
20   /// The key may be any borrowed form of the map's key type,
21   /// but `Hash` and `Eq` on the borrowed form must match those for the key.
22   pub fn get_mut<Q>(&mut self, key: &Q) -> Option<&&mut V> {/* --snip-- */}
23
24   /// Returns `true` if there is an entry corresponding to the key in the map.
25   pub fn contains_key<Q>(&self, key: &Q) -> bool {/* --snip-- */}
26
27   /// Converts the OccupiedEntry into a mutable reference to the value in the map
28   /// with a lifetime bound to the map itself.
29   pub fn into_mut(self) -> &'a mut V {/* --snip-- */}
30
31   /// Gets the given key's corresponding entry in the map for in-place modification.
32   pub fn entry(&mut self, key: K) -> Entry<K, V> {/* --snip-- */}Copy to clipboard

```

Initializing a HashMap

As mentioned, not initializing storage before you use it is a common error that can break your smart contract. For each key in a storage value, the value needs to be set before you can use it. To do this, we will create a private function which handles when the value is set and when it is not, and make sure we never work with uninitialized storage.

So given `my_number_map`, imagine we wanted the default value for any given key to be `0`. We can build a function like this:

```

1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]

```

```

6 mod mycontract {
7
8     #[ink(storage)]
9     pub struct MyContract {
10         // Store a mapping from AccountIds to a u32
11         my_number_map: ink_storage::collections::HashMap<AccountId, u32>,
12     }
13
14     impl MyContract {
15         /// Private function.
16         /// Returns the number for an AccountId or 0 if it is not set.
17         fn my_number_or_zero(&self, of: &AccountId) -> u32 {
18             let balance = self.my_number_map.get(of).unwrap_or(&0);
19             *balance
20         }
21     }
22 }
```

Here we see that after we `get` the value from `my_number_map` we call `unwrap_or` which will either `unwrap` the value stored in storage, *or* if there is no value, return some known value. Then, when building functions that interact with this HashMap, you need to always remember to call this function rather than getting the value directly from storage.

Here is an example:

```

1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod mycontract {
7
8      #[ink(storage)]
9      pub struct MyContract {
10          // Store a mapping from AccountIds to a u32
11          my_number_map: ink_storage::collections::HashMap<AccountId, u32>,
12      }
13
14      impl MyContract {
15          // Get the value for a given AccountId
16          #[ink(message)]
17          pub fn get(&self, of: AccountId) -> u32 {
18              self.my_number_or_zero(&of)
19          }
20      }
```

```

21     // Get the value for the calling AccountId
22     #[ink(message)]
23     pub fn get_my_number(&self) -> u32 {
24         let caller = self.env().caller();
25         self.my_number_or_zero(&caller)
26     }
27
28     // Returns the number for an AccountId or 0 if it is not set.
29     fn my_number_or_zero(&self, of: &AccountId) -> u32 {
30         let value = self.my_number_map.get(of).unwrap_or(&0);
31         *value
32     }
33 }
34 }
```

Contract Caller

As you might have noticed in the example above, we use a special function called `self.env().caller()`. This function is available throughout the contract logic and will always return to you the contract caller.

NOTE: The contract caller is not the same as the origin caller. If a user triggers a contract which then calls a subsequent contract, the `self.env().caller()` in the second contract will be the address of the first contract, not the original user.

`self.env().caller()` can be used a number of different ways. In the examples above, we are basically creating an "access control" layer which allows a user to modify their own value, but no one else. You can also do things like define a contract owner during contract deployment:

```

1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod mycontract {
7
8      #[ink(storage)]
9      pub struct MyContract {
10          // Store a contract owner
11          owner: AccountId,
12      }
```

```
13     impl MyContract {
14         #[ink(constructor)]
15         pub fn new(init_value: i32) -> Self {
16             Self {
17                 owner: Self::env().caller();
18             }
19         }
20         /* --snip-- */
21     }
22 }
```

Then you can write permissioned functions which checks that the current caller is the owner of the contract.

Your Turn!

Follow the `ACTION`s in the template code to introduce a storage map to your contract.

Remember to run `cargo +nightly test` to test your work.

□ Starting Point

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod incrementer {
7
8      #[ink(storage)]
9      pub struct Incrementer {
10          value: i32,
11          // ACTION: Add a `HashMap` from `AccountId` to `u64` named `my_value`
12      }
13
14      impl Incrementer {
15          #[ink(constructor)]
16          pub fn new(init_value: i32) -> Self {
17              Self {
18                  value: init_value,
19              }
20          }
21
22          #[ink(constructor)]
23          pub fn default() -> Self {
24              Self {
25                  value: 0,
26              }
27          }
28
29          #[ink(message)]
30          pub fn get(&self) -> i32 {
31              self.value
32          }
33
34          #[ink(message)]
35          pub fn inc(&mut self, by: i32) {
36              self.value += by;
37          }
38
39          #[ink(message)]
40          pub fn get_mine(&self) -> u64 {
41              // ACTION: Get `my_value` using `my_value_or_zero` on `&self`
42              // ACTION: Return `my_value`
43          }
44
45          fn my_value_or_zero(&self, of: &AccountId) -> u64 {
46              // ACTION: `get` and return the value of `of` and `unwrap`
47          }
}
```

```
48     }
49
50     #[cfg(test)]
51     mod tests {
52         use super::*;

53         // Alias `ink_lang` so we can use `ink::test`.
54         use ink_lang as ink;
55
56         #[test]
57         fn default_works() {
58             let contract = Incrementer::default();
59             assert_eq!(contract.get(), 0);
60         }
61
62         #[test]
63         fn it_works() {
64             let mut contract = Incrementer::new(42);
65             assert_eq!(contract.get(), 42);
66             contract.inc(5);
67             assert_eq!(contract.get(), 47);
68             contract.inc(-50);
69             assert_eq!(contract.get(), -3);
70         }
71
72         // Use `ink::test` to initialize accounts.
73         #[ink::test]
74         fn my_value_works() {
75             let contract = Incrementer::new(11);
76             assert_eq!(contract.get(), 11);
77             assert_eq!(contract.get_mine(), 0);
78         }
79     }
80 }
81 }
```

✓Potential Solution

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod incrementer {
7
8      #[ink(storage)]
9      pub struct Incrementer {
10          value: i32,
11          my_value: ink_storage::collections::HashMap<AccountId, u64>,
12      }
13
14     impl Incrementer {
15         #[ink(constructor)]
16         pub fn new(init_value: i32) -> Self {
17             Self {
18                 value: init_value,
19                 my_value: ink_storage::collections::HashMap::new(),
20             }
21         }
22
23         #[ink(constructor)]
24         pub fn default() -> Self {
25             Self {
26                 value: 0,
27                 my_value: Default::default(),
28             }
29         }
30
31         #[ink(message)]
32         pub fn get(&self) -> i32 {
33             self.value
34         }
35
36         #[ink(message)]
37         pub fn inc(&mut self, by: i32) {
38             self.value += by;
39         }
40
41         #[ink(message)]
42         pub fn get_mine(&self) -> u64 {
43             self.my_value_or_zero(&self.env().caller())
44         }
45
46         fn my_value_or_zero(&self, of: &AccountId) -> u64 {
47             *self.my_value.get(of).unwrap_or(&0)
```

```
48         }
49     }
50
51     #[cfg(test)]
52     mod tests {
53         use super::*;
54
55         use ink_lang as ink;
56
57         #[test]
58         fn default_works() {
59             let contract = Incrementer::default();
60             assert_eq!(contract.get(), 0);
61         }
62
63         #[test]
64         fn it_works() {
65             let mut contract = Incrementer::new(42);
66             assert_eq!(contract.get(), 42);
67             contract.inc(5);
68             assert_eq!(contract.get(), 47);
69             contract.inc(-50);
70             assert_eq!(contract.get(), -3);
71         }
72
73         #[ink::test]
74         fn my_value_works() {
75             let contract = Incrementer::new(11);
76             assert_eq!(contract.get(), 11);
77             assert_eq!(contract.get_mine(), 0);
78         }
79     }
80 }
```

Incrementing My Value

The final step in our Incrementer contract is to allow each user to update increment their own value.

Modifying a HashMap

Making changes to the value of a HashMap is just as sensitive as getting the value. If you try to modify some value before it has been initialized, your contract will panic!

But have no fear, we can continue to use the `my_number_or_zero` function we created to protect us from these situations!

```
1  impl MyContract {
2
3      /* --snip-- */
4
5      // Set the value for the calling AccountId
6      #[ink(message)]
7      pub fn set_my_number(&mut self, value: u32) {
8          let caller = self.env().caller();
9          self.my_number_map.insert(caller, value);
10     }
11
12     // Add a value to the existing value for the calling AccountId
13     #[ink(message)]
14     pub fn add_my_number(&mut self, value: u32) {
15         let caller = self.env().caller();
16         let my_number = self.my_number_or_zero(&caller);
17         self.my_number_map.insert(caller, my_number + value);
18     }
19
20     /// Returns the number for an AccountId or 0 if it is not set.
21     fn my_number_or_zero(&self, of: &AccountId) -> u32 {
22         *self.my_number_map.get(of).unwrap_or(&0)
23     }
24 }
```

Here we have written two kinds of functions which modify a HashMap. One which simply inserts the value directly into storage, with no need to read the value first, and the other which modifies the existing value. Note how we can always `insert` the value without worry, as that initialized the value in storage, but before you can get or modify anything, we need to call `my_number_or_zero` to make sure we are working with a real value.

Feel the Pain (Optional)

We will not always have an existing value on our contract's storage. We can take advantage of the Rust `Option<T>` type to help use on this task. If there's no value on the contract storage we will insert a new one; on the contrary if there is an existing value we will only update it.

ink! HashMaps expose the well-known `entry` API that we can use to achieve this type of "upset" behavior:

```
1 let caller = self.env().caller();
2 self.my_number_map
3     .entry(caller)
4     .and_modify(|old_value| old_value += by)
5     .or_insert(by);
```

Your Turn!

Follow the `ACTION`s to finish your Incrementer smart contract.

Remember to run `cargo +nightly test` to test your work.

□ Starting Point

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod incrementer {
7      #[ink(storage)]
8      pub struct Incrementer {
9          value: i32,
10         my_value: ink_storage::collections::HashMap<AccountId, u64>,
11     }
12
13     impl Incrementer {
14         #[ink(constructor)]
15         pub fn new(init_value: i32) -> Self {
16             Self {
17                 value: init_value,
18                 my_value: ink_storage::collections::HashMap::new(),
19             }
20         }
21
22         #[ink(constructor)]
23         pub fn default() -> Self {
24             Self {
25                 value: 0,
26                 my_value: Default::default(),
27             }
28         }
29
30         #[ink(message)]
31         pub fn get(&self) -> i32 {
32             self.value
33         }
34
35         #[ink(message)]
36         pub fn inc(&mut self, by: i32) {
37             self.value += by;
38         }
39
40         #[ink(message)]
41         pub fn get_mine(&self) -> u64 {
42             self.my_value_or_zero(&self.env().caller())
43         }
44
45         #[ink(message)]
46         pub fn inc_mine(&mut self, by: u64) {
47             // ACTION: Get the `caller` of this function.
```

```
48         // ACTION: Get `my_value` that belongs to `caller` by using
49         // ACTION: Insert the incremented `value` back into the map
50     }
51
52     fn my_value_or_zero(&self, of: &AccountId) -> u64 {
53         *self.my_value.get(of).unwrap_or(&0)
54     }
55 }
56
57 #[cfg(test)]
58 mod tests {
59     use super::*;


```

✓Potential Solution

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod incrementer {
7      #[ink(storage)]
8      pub struct Incrementer {
9          value: i32,
10         my_value: ink_storage::collections::HashMap<AccountId, u64>,
11     }
12
13     impl Incrementer {
14         #[ink(constructor)]
15         pub fn new(init_value: i32) -> Self {
16             Self {
17                 value: init_value,
18                 my_value: ink_storage::collections::HashMap::new(),
19             }
20         }
21
22         #[ink(constructor)]
23         pub fn default() -> Self {
24             Self {
25                 value: 0,
26                 my_value: Default::default(),
27             }
28         }
29
30         #[ink(message)]
31         pub fn get(&self) -> i32 {
32             self.value
33         }
34
35         #[ink(message)]
36         pub fn inc(&mut self, by: i32) {
37             self.value += by;
38         }
39
40         #[ink(message)]
41         pub fn get_mine(&self) -> u64 {
42             self.my_value_or_zero(&self.env().caller())
43         }
44
45         #[ink(message)]
46         pub fn inc_mine(&mut self, by: u64) {
47             let caller = self.env().caller();
```

```
48         let my_value = self.my_value_or_zero(&caller);
49         self.my_value.insert(caller, my_value + by);
50     }
51
52     fn my_value_or_zero(&self, of: &AccountId) -> u64 {
53         *self.my_value.get(of).unwrap_or(&0)
54     }
55 }
56
57 #[cfg(test)]
58 mod tests {
59     use super::*;


```

WASM ERC20

Introduction

In this chapter, we will show you how you can build an ERC20 token contract with ink!.

Over the course of the chapter, we will cover:

- Initial token minting
- Token transfers
- Approvals and third party transfers
- Emitting runtime events through Substrate

But first, we will go over the ERC20 standard for those of you who are not familiar.

ERC20 Standard

The [ERC20 token standard](#) defines the interface for the most popular Ethereum smart contract.

```
1 // -----
2 // ERC Token Standard #20 Interface
3 // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
4 // -----
5
6 contract ERC20Interface {
7     // Storage Getters
8     function totalSupply() public view returns (uint);
9     function balanceOf(address tokenOwner) public view returns (uint balance);
10    function allowance(address tokenOwner, address spender) public view returns (uint);
11
12    // Public Functions
13    function transfer(address to, uint tokens) public returns (bool success);
14    function approve(address spender, uint tokens) public returns (bool success);
15    function transferFrom(address from, address to, uint tokens) public returns (bool success);
16
17    // Contract Events
18    event Transfer(address indexed from, address indexed to, uint tokens);
19    event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
20 }
```

In summary, it allows individuals to deploy their own cryptocurrency on top of an existing smart contract platform. There isn't much magic happening in this contract. Users balances are stored in a HashMap, and a set of APIs are built to allow users to transfer tokens they own or allow a third party to transfer some amount of tokens on their behalf. Most importantly, all of this logic is implemented ensuring that funds are not unintentionally created or destroyed, and that a user's funds are protected from malicious actors.

Note that all the public functions return a `bool` which specifies if the call was successful or not. We will adhere to that specification.

Creating the ERC20 Template

We are going to start another ink! project to build an ERC20 token contract.

Back in your working directory, run:

```
cargo contract new erc20
```

Again, we will replace the `lib.rs` file content with the template provided on this page.

You will notice that the template for the ERC20 token is VERY similar to the Incrementer contract. (Coincidence? ¯_(ツ)_/¯)

The storage (so far) consists of:

- A storage `Value` : representing the total supply of tokens in our contract.
- A storage `HashMap` : representing the individual balance of each account.

ERC20 Deployment

The most basic ERC20 token contract is a fixed supply token. During contract deployment, all the tokens will be automatically given to the contract creator. It is then up to that user to distribute those tokens to other users as they see fit.

Of course, this is not the only way to mint and distribute tokens, but the most simple one, and what we will be doing here.

So remember to `set` the total balance and `insert` the balance of the `Self::env().caller()`

Your Turn!

This chapter should be nothing more than a quick refresher of the content you already learned.

You need to:

- Set up a constructor function which initializes the two storage items
- Create getters for both storage items
- Create a `balance_of_or_zero` function to handle reading values from the HashMap

Remember to run `cargo +nightly test` to test your work.

Starting Point

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod erc20 {
7      #[cfg(not(feature = "ink-as-dependency"))]
8      #[ink(storage)]
9      pub struct Erc20 {
10          /// The total supply.
11          total_supply: Balance,
12          /// The balance of each user.
13          balances: ink_storage::collections::HashMap<AccountId, Balance>
14      }
15
16      impl Erc20 {
17          #[ink(constructor)]
18          pub fn new(initial_supply: Balance) -> Self {
19              // ACTION: `set` the total supply to `init_value`
20              // ACTION: `insert` the `init_value` as the `caller` balance
21          }
22
23          #[ink(message)]
24          pub fn total_supply(&self) -> Balance {
25              // ACTION: Return the total supply
26          }
27
28          #[ink(message)]
29          pub fn balance_of(&self, owner: AccountId) -> Balance {
30              // ACTION: Return the balance of `owner`
31              // HINT: Use `balance_of_or_zero` to get the `owner` balance
32          }
33
34          fn balance_of_or_zero(&self, owner: &AccountId) -> Balance {
35              // ACTION: `get` the balance of `owner`, then `unwrap_or`
36              // ACTION: Return the balance
37          }
38      }
39
40      #[cfg(test)]
41      mod tests {
42          use super::*;

43          use ink_lang as ink;
44
45          #[ink::test]
46          fn new_works() {
```

```
48         let contract = Erc20::new(777);
49         assert_eq!(contract.total_supply(), 777);
50     }
51
52 #[ink::test]
53 fn balance_works() {
54     let contract = Erc20::new(100);
55     assert_eq!(contract.total_supply(), 100);
56     assert_eq!(contract.balance_of(AccountId::from([0x1; 32]));
57     assert_eq!(contract.balance_of(AccountId::from([0x0; 32]));
58 }
59 }
60 }
```

✓Potential Solution

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod erc20 {
7      #[cfg(not(feature = "ink-as-dependency"))]
8      #[ink(storage)]
9      pub struct Erc20 {
10          /// The total supply.
11          total_supply: Balance,
12          /// The balance of each user.
13          balances: ink_storage::collections::HashMap<AccountId, Balance>
14      }
15
16      impl Erc20 {
17          #[ink(constructor)]
18          pub fn new(initial_supply: Balance) -> Self {
19              let mut balances = ink_storage::collections::HashMap::new();
20              balances.insert(Self::env().caller(), initial_supply);
21              Self {
22                  total_supply: initial_supply,
23                  balances
24              }
25          }
26
27          #[ink(message)]
28          pub fn total_supply(&self) -> Balance {
29              self.total_supply
30          }
31
32          #[ink(message)]
33          pub fn balance_of(&self, owner: AccountId) -> Balance {
34              self.balance_of_or_zero(&owner)
35          }
36
37          fn balance_of_or_zero(&self, owner: &AccountId) -> Balance {
38              *self.balances.get(owner).unwrap_or(&0)
39          }
40      }
41
42      #[cfg(test)]
43      mod tests {
44          use super::*;

45          use ink_lang as ink;
46
47      }
```

```
48     #[ink::test]
49     fn new_works() {
50         let contract = Erc20::new(777);
51         assert_eq!(contract.total_supply(), 777);
52     }
53
54     #[ink::test]
55     fn balance_works() {
56         let contract = Erc20::new(100);
57         assert_eq!(contract.total_supply(), 100);
58         assert_eq!(contract.balance_of(AccountId::from([0x1; 32]));
59         assert_eq!(contract.balance_of(AccountId::from([0x0; 32])));
60     }

```

Transferring Tokens

So at this point, we have a single user that owns all the tokens for the contract. However, it's not *really* a useful token unless you can transfer them to other people...

Let's do that!

The Transfer Function

The `transfer` function does exactly what you might expect: it allows the user calling the contract to transfer some funds they own to another user.

You will notice in our template code there is a public function `transfer` and an internal function `transfer_from_to`. We have done this because in the future, we will be reusing the logic for a token transfer when we enable third party allowances and spending on-behalf-of.

`transfer_from_to()`

```
fn transfer_from_to(&mut self, from: AccountId, to: AccountId, value: Balance)
```

The `transfer_from_to` function will be built without any *authorization* checks. Because it is an internal function, we fully control when it gets called. However, it will have all logical checks around managing the balances between accounts.

Really we just need to check for one thing: make sure that the `from` account has enough funds to send to the `to` account:

```
1 if balance_from < value {
2     return false
3 }
```

Remember that the `transfer` function and other public functions return a bool to indicate success. If the `from` account does not have enough balance to satisfy the transfer, we will exit early and return false, not making any changes to the contract state. Our `transfer_from_to` will simply forward the "success" bool up to the function that calls it.

transfer()

```
1 #[ink(message)]
2 pub fn transfer(&mut self, to: AccountId, value: Balance) -> bool {/* --snip-- */}
```

Finally, the `transfer` function will simply call into the `transfer_from_to` with the `from` parameter automatically set to the `self.env().caller()`. This is our "authorization check" since the contract caller is always authorized to move their own funds.

Transfer Math

There really is not much to say about the simple math executed within a token transfer.

1. First we get the current balance of both the `from` and `to` account, making sure to use our `balance_of_or_zero()` getter.
 2. Then we make the logic check mentioned above to ensure the `from` balance has enough funds to send `value`.
 3. Finally, we subtract that `value` from the `from` balance and add it to the `to` balance and insert those new values back in.
-

Your Turn!

Follow the `ACTION`s in the template code to build your transfer function.

Remember to run `cargo +nightly test` to test your work.

Starting Point

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod erc20 {
7      #[cfg(not(feature = "ink-as-dependency"))]
8      #[ink(storage)]
9      pub struct Erc20 {
10          /// The total supply.
11          total_supply: Balance,
12          /// The balance of each user.
13          balances: ink_storage::collections::HashMap<AccountId, Balance>
14      }
15
16      impl Erc20 {
17          #[ink(constructor)]
18          pub fn new(initial_supply: Balance) -> Self {
19              let mut balances = ink_storage::collections::HashMap::new();
20              balances.insert(Self::env().caller(), initial_supply);
21              Self {
22                  total_supply: initial_supply,
23                  balances
24              }
25          }
26
27          #[ink(message)]
28          pub fn total_supply(&self) -> Balance {
29              self.total_supply
30          }
31
32          #[ink(message)]
33          pub fn balance_of(&self, owner: AccountId) -> Balance {
34              self.balance_of_or_zero(&owner)
35          }
36
37          #[ink(message)]
38          pub fn transfer(&mut self, to: AccountId, value: Balance) -> Balance {
39              // ACTION: Call the `transfer_from_to` with `from` as `self`
40          }
41
42          fn transfer_from_to(&mut self, from: AccountId, to: AccountId) -> Balance {
43              // ACTION: Get the balance for `from` and `to`
44              // HINT: Use the `balance_of_or_zero` function to do this
45              // ACTION: If `from_balance` is less than `value`, return Err("Insufficient funds")
46              // ACTION: Insert new values for `from` and `to`
47              //           * from_balance - value
48          }
49
50      }
51
52      #[ink(message)]
53      pub fn transfer_from_to(&self, from: AccountId, to: AccountId) -> Balance {
54          // ACTION: Get the balance for `from` and `to`
55          // HINT: Use the `balance_of_or_zero` function to do this
56          // ACTION: If `from_balance` is less than `value`, return Err("Insufficient funds")
57          // ACTION: Insert new values for `from` and `to`
58          //           * from_balance - value
59      }
60
61      #[ink(message)]
62      pub fn approve(&self, owner: AccountId, spender: AccountId, value: Balance) -> Balance {
63          // ACTION: Call the `approve` function on the `spender` account
64          // HINT: Use the `balance_of` function to get the current allowance
65          // ACTION: Insert new values for `owner` and `spender`
66          //           * allowance + value
67      }
68
69      #[ink(message)]
70      pub fn increase_allowance(&self, spender: AccountId, value: Balance) -> Balance {
71          // ACTION: Call the `increase_allowance` function on the `spender` account
72          // HINT: Use the `balance_of` function to get the current allowance
73          // ACTION: Insert new values for `spender` and `allowance`
74          //           * allowance + value
75      }
76
77      #[ink(message)]
78      pub fn decrease_allowance(&self, spender: AccountId, value: Balance) -> Balance {
79          // ACTION: Call the `decrease_allowance` function on the `spender` account
80          // HINT: Use the `balance_of` function to get the current allowance
81          // ACTION: Insert new values for `spender` and `allowance`
82          //           * allowance - value
83      }
84
85      #[ink(message)]
86      pub fn transfer_from(&self, from: AccountId, to: AccountId, value: Balance) -> Balance {
87          // ACTION: Call the `transfer` function on the `from` account
88          // HINT: Use the `balance_of` function to get the current balance of `from`
89          // ACTION: Insert new values for `from` and `to`
90          //           * from_balance - value
91      }
92
93      #[ink(message)]
94      pub fn transfer_to(&self, from: AccountId, to: AccountId, value: Balance) -> Balance {
95          // ACTION: Call the `transfer` function on the `to` account
96          // HINT: Use the `balance_of` function to get the current balance of `to`
97          // ACTION: Insert new values for `from` and `to`
98          //           * to_balance + value
99      }
100 }
```

```
48          //           * to_balance + value
49          // ACTION: Return `true`
50      }
51
52  fn balance_of_or_zero(&self, owner: &AccountId) -> Balance {
53      *self.balances.get(owner).unwrap_or(&0)
54  }
55 }
56
57 #[cfg(test)]
58 mod tests {
59     use super::*;


```

✓Potential Solution

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod erc20 {
7      #[cfg(not(feature = "ink-as-dependency"))]
8      #[ink(storage)]
9      pub struct Erc20 {
10          /// The total supply.
11          total_supply: Balance,
12          /// The balance of each user.
13          balances: ink_storage::collections::HashMap<AccountId, Balance>;
14      }
15
16      impl Erc20 {
17          #[ink(constructor)]
18          pub fn new(initial_supply: Balance) -> Self {
19              let mut balances = ink_storage::collections::HashMap::new();
20              balances.insert(Self::env().caller(), initial_supply);
21              Self {
22                  total_supply: initial_supply,
23                  balances
24              }
25          }
26
27          #[ink(message)]
28          pub fn total_supply(&self) -> Balance {
29              self.total_supply
30          }
31
32          #[ink(message)]
33          pub fn balance_of(&self, owner: AccountId) -> Balance {
34              self.balance_of_or_zero(&owner)
35          }
36
37          #[ink(message)]
38          pub fn transfer(&mut self, to: AccountId, value: Balance) -> Result<(), ()> {
39              self.transfer_from_to(self.env().caller(), to, value)
40          }
41
42          fn transfer_from_to(&mut self, from: AccountId, to: AccountId, value: Balance) {
43              let from_balance = self.balance_of_or_zero(&from);
44              if from_balance < value {
45                  return false
46              }
47          }
48      }
49  }
```

```
48     // Update the sender's balance.
49     self.balances.insert(from, from_balance - value);
50
51     // Update the receiver's balance.
52     let to_balance = self.balance_of_or_zero(&to);
53     self.balances.insert(to, to_balance + value);
54
55     true
56 }
57
58 fn balance_of_or_zero(&self, owner: &AccountId) -> Balance {
59     *self.balances.get(owner).unwrap_or(&0)
60 }
```

Creating an Event

Recall that contract calls cannot directly return a value to the outside world when submitting a transaction. However, often we will want to indicate to the outside world that something has taken place (e.g. a transaction has occurred or a certain state has been reached). We can alert others that this has occurred using an `event`.

Declaring Events

An event can communicate an arbitrary amount of data, defined in a similar manner as a `struct`. Events should be declared using the `#[ink(event)]` attribute.

For example,

```
1 #[ink(event)]
2 pub struct Transfer {
3     #[ink(topic)]
4     from: Option<AccountId>,
5     #[ink(topic)]
6     to: Option<AccountId>,
7     value: Balance,
8 }
```

This `Transfer` event will contain three pieces of data - a value of type `Balance` and two Option-wrapped `AccountId` variables indicating the `from` and `to` accounts. For faster access to the event data they can have *indexed fields*. We can do this by using the `#[ink(topic)]` attribute tag on that field.

One way of retrieving data from an Option variable is using the `.unwrap_or()` function. You may recall using this in the `my_value_or_zero()` and `balance_of_or_zero()` functions in this project and the Incrementer project.

Emitting Events

Now that we have defined what data will be contained within the event and how to declare it, it's time to actually emit some events. We do this by calling `self.env().emit_event` and include an event as the sole argument to the method call.

Remember that since the `from` and `to` fields are Option, we can't just set them to particular values. Let's assume we want to set an value of 100 for the initial deployer. This value does not come from any other account, and so the `from` value should be `None`.

```
1 self.env()  
2     .emit_event(  
3         Transfer {  
4             from: None,  
5             to: Some(self.env().caller()),  
6             value: 100,  
7         });
```



Note: `value` does not need a `Some()`, as the value is not stored in an `Option`.

We want to emit a Transfer event every time that a transfer takes place. In the ERC-20 template that we have been working on, this occurs in two places: first, during the `new` call, and second, every time that `transfer_from_to` is called.

Your Turn!

Follow the ACTIONS in the template code to emit a `Transfer` event every time a token transfer occurs.

Remember to run `cargo +nightly test` to test your work.

Starting Point

```
1  ![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod erc20 {
7      #[cfg(not(feature = "ink-as-dependency"))]
8      #[ink(storage)]
9      pub struct Erc20 {
10         /// The total supply.
11         total_supply: Balance,
12         /// The balance of each user.
13         balances: ink_storage::collections::HashMap<AccountId, Balance>,
14     }
15
16     #[ink(event)]
17     pub struct Transfer {
18         // ACTION: Create a `Transfer` event with:
19         //           * from: Option<AccountId>
20         //           * to: Option<AccountId>
21         //           * value: Balance
22     }
23
24     impl Erc20 {
25         #[ink(constructor)]
26         pub fn new(initial_supply: Balance) -> Self {
27             let mut balances = ink_storage::collections::HashMap::new();
28             balances.insert(Self::env().caller(), initial_supply);
29
30             // ACTION: Call `self.env().emit_event` with the `Transfer` event
31             // HINT: Since we use `Option<AccountId>`, you need to use
32
33             Self {
34                 total_supply: initial_supply,
35                 balances
36             }
37         }
38
39         #[ink(message)]
40         pub fn total_supply(&self) -> Balance {
41             self.total_supply
42         }
43
44         #[ink(message)]
45         pub fn balance_of(&self, owner: AccountId) -> Balance {
46             self.balance_of_or_zero(&owner)
47         }
48     }
49 }
```

```
48
49     #[ink(message)]
50     pub fn transfer(&mut self, to: AccountId, value: Balance) -> I
51         self.transfer_from_to(self.env().caller(), to, value)
52     }
53
54     fn transfer_from_to(&mut self, from: AccountId, to: AccountId)
55         let from_balance = self.balance_of_or_zero(&from);
56         if from_balance < value {
57             return false
58         }
59
60         // Update the sender's balance.
```

✓Potential Solution

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod erc20 {
7      #[cfg(not(feature = "ink-as-dependency"))]
8      #[ink(storage)]
9      pub struct Erc20 {
10          /// The total supply.
11          total_supply: Balance,
12          /// The balance of each user.
13          balances: ink_storage::collections::HashMap<AccountId, Balance>;
14      }
15
16      #[ink(event)]
17      pub struct Transfer {
18          #[ink(topic)]
19          from: Option<AccountId>,
20          #[ink(topic)]
21          to: Option<AccountId>,
22          #[ink(topic)]
23          value: Balance,
24      }
25
26      impl Erc20 {
27          #[ink(constructor)]
28          pub fn new(initial_supply: Balance) -> Self {
29              let caller = Self::env().caller();
30              let mut balances = ink_storage::collections::HashMap::new();
31              balances.insert(caller, initial_supply);
32
33              Self::env().emit_event(Transfer {
34                  from: None,
35                  to: Some(caller),
36                  value: initial_supply,
37              });
38
39              Self {
40                  total_supply: initial_supply,
41                  balances
42              }
43          }
44
45          #[ink(message)]
46          pub fn total_supply(&self) -> Balance {
47              self.total_supply
```

```

48     }
49
50     #[ink(message)]
51     pub fn balance_of(&self, owner: AccountId) -> Balance {
52         self.balance_of_or_zero(&owner)
53     }
54
55     #[ink(message)]
56     pub fn transfer(&mut self, to: AccountId, value: Balance) -> Balance {
57         self.transfer_from_to(self.env().caller(), to, value)
58     }
59
60     fn transfer_from_to(&mut self, from: AccountId, to: AccountId) -> Balance {
61         let from_balance = self.balance_of_or_zero(&from);
62         if from_balance < value {
63             return false
64         }
65
66         // Update the sender's balance.
67         self.balances.insert(from, from_balance - value);
68
69         // Update the receiver's balance.
70         let to_balance = self.balance_of_or_zero(&to);
71         self.balances.insert(to, to_balance + value);
72
73         self.env().emit_event(Transfer {
74             from: Some(from),
75             to: Some(to),
76             value,
77         });
78
79         true
80     }
81
82     fn balance_of_or_zero(&self, owner: &AccountId) -> Balance {
83         *self.balances.get(owner).unwrap_or(&0)
84     }
85 }
86
87 #[cfg(test)]
88 mod tests {
89     use super::*;

90
91     use ink_lang as ink;
92
93     #[ink::test]
94     fn new_works() {
95         let contract = Erc20::new(777);
96         assert_eq!(contract.total_supply(), 777);
97     }
98
99     #[ink::test]

```

```
100     fn balance_works() {
101         let contract = Erc20::new(100);
102         assert_eq!(contract.total_supply(), 100);
103         assert_eq!(contract.balance_of(AccountId::from([0x1; 32]))
104             assert_eq!(contract.balance_of(AccountId::from([0x0; 32]))
105         }
106
107 #[ink::test]
108 fn transfer_works() {
109     let mut contract = Erc20::new(100);
110     assert_eq!(contract.balance_of(AccountId::from([0x1; 32]))
111     assert!(contract.transfer(AccountId::from([0x0; 32]), 10)
112     assert_eq!(contract.balance_of(AccountId::from([0x0; 32]))
113     assert!(!contract.transfer(AccountId::from([0x0; 32]), 10
114         )
115     }
116 }
```

Supporting Approvals and Transfer From

We are almost there! Our token contract can now transfer funds from user to user and tell the outside world what is going on when this happens. All that is left to do is introduce the `approve` and `transfer_from` functions.

Third Party Transfers

This section is all about adding the ability for other accounts to safely spend some amount of your tokens.

The immediate question should be: "Why the heck would I want that?"

Well, one such scenario is to support Decentralized Exchanges. Basically, other smart contracts can allow you to exchange tokens with other users, usually one type of token for another. However, these "bids" do not always execute right away. Maybe you want to get a really good deal for token trade, and will hold out until that trade is met.

Well, rather than giving your tokens directly to the contract (an escrow), you can simply "approve" them to spend some of your tokens on your behalf! This means that during the time while you are waiting for a trade to execute, you can still control and spend your funds if needed. Better yet, you can approve multiple different contracts or users to access your funds, so if one contract offers the best trade, you do not need to pull out funds from the other and move them, a sometimes costly and time consuming process.

So hopefully you can see why a feature like this would be useful, but how can we do it safely?

We use a two step process: **Approve** and **Transfer From**.

Approve

Approving another account to spend your funds is the first step in the third party transfer process. A token owner can specify another account and any arbitrary number of tokens it can spend on the owner's behalf. The owner need not have all their funds approved to be

spent by others; in the situation where there is not enough funds, the approved account can spend up to the approved amount from the owner's balance.

When an account calls `approve` multiple times, the approved value simply overwrites any existing value that was approved in the past. By default, the approved value between any two accounts is `0`, and a user can always call `approve` for `0` to revoke access to their funds from another account.

To store approvals in our contract, we need to use a slightly fancy `HashMap`.

Since each account can have a different amount approved for any other account to use, we need to use a tuple as our key which simply points to a balance value. Here is an example of what that would look like:

```
1 pub struct Erc20 {  
2     /// Balances that are spendable by non-owners: (owner, spender) -> allowance  
3     allowances: ink_storage::collections::HashMap<(AccountId, AccountId), Balance>  
4 }
```

Here we have defined the tuple to represent `(owner, spender)` such that we can look up how much a "spender" can spend from an "owner's" balance using the `AccountId`s in this tuple. Remember that we will need to again create an `allowance_of_or_zero` function to help us get the allowance of an account when it is not initialized, and a getter function called `allowance` to look up the current value for any pair of accounts.

```
1 /// Approve the passed AccountId to spend the specified amount of tokens  
2 /// on the behalf of the message's sender.  
3 #[ink(message)]  
4 pub fn approve(&mut self, spender: AccountId, value: Balance) -> bool /* - */
```

When you call the `approve` function, you simply insert the `value` specified into storage. The `owner` is always the `self.env().caller()`, ensuring that the function call is always authorized.

Transfer From

Finally, once we have set up an approval for one account to spend on-behalf-of another, we need to create a special `transfer_from` function which enables an approved user to transfer those funds.

As mentioned earlier, we will take advantage of the private `transfer_from_to` function to do the bulk of our transfer logic. All we need to introduce is the *authorization* logic again.

So what does it mean to be authorized to call this function?

1. The `self.env().caller()` must have some allowance to spend funds from the `from` account.
2. The allowance must not be less than the value trying to be transferred.

In code, that can easily be represented like so:

```
1 let allowance = self.allowance_of_or_zero(&from, &self.env().caller());
2 if allowance < value {
3     return false
4 }
5 /* --snip-- */
6 true
```

Again, we exit early and return false if our authorization does not pass.

If everything looks good though, we simply `insert` the updated allowance into the `allowance` `HashMap` (`let new_allowance = allowance - value`), and call the `transfer_from_to` between the specified `from` and `to` accounts.

Be Careful!

If you glaze over the logic of this function too quickly, you may introduce a bug into your smart contract. Remember when calling `transfer_from`, the `self.env().caller()` and

the `from` account is used to look up the current allowance, but the `transfer_from` function is called between the `from` and `to` account specified.

There are three account variables in play whenever `transfer_from` is called, and you need to make sure to use them correctly! Hopefully our test will catch any mistake you make.

Your Turn!

You are almost there! This is the last piece of the ERC20 token contract.

Follow the `ACTION`'s in the contract template to finish your ERC20 implementation.

Remember to run `cargo +nightly test` to test your work.

Starting Point

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod erc20 {
7      #[cfg(not(feature = "ink-as-dependency"))]
8      #[ink(storage)]
9      pub struct Erc20 {
10          /// The total supply.
11          total_supply: Balance,
12          /// The balance of each user.
13          balances: ink_storage::collections::HashMap<AccountId, Balance>,
14          /// Approval spender on behalf of the message's sender.
15          allowances: ink_storage::collections::HashMap<(AccountId, AccountId), Balance>,
16      }
17
18      #[ink(event)]
19      pub struct Transfer {
20          #[ink(topic)]
21          from: Option<AccountId>,
22          #[ink(topic)]
23          to: Option<AccountId>,
24          #[ink(topic)]
25          value: Balance,
26      }
27
28      #[ink(event)]
29      pub struct Approval {
30          #[ink(topic)]
31          owner: AccountId,
32          #[ink(topic)]
33          spender: AccountId,
34          #[ink(topic)]
35          value: Balance,
36      }
37
38      impl Erc20 {
39          #[ink(constructor)]
40          pub fn new(initial_supply: Balance) -> Self {
41              let caller = Self::env().caller();
42              let mut balances = ink_storage::collections::HashMap::new();
43              balances.insert(caller, initial_supply);
44
45              Self::env().emit_event(Transfer {
46                  from: None,
47                  to: Some(caller),
```

```
48             value: initial_supply,
49         });
50
51     Self {
52         total_supply: initial_supply,
53         balances,
54         allowances: ink_storage::collections::HashMap::new(),
55     }
56 }
57
58 #[ink(message)]
59 pub fn total_supply(&self) -> Balance {
60     self.total_supply
61 }
62
63 #[ink(message)]
64 pub fn balance_of(&self, owner: AccountId) -> Balance {
65     self.balance_of_or_zero(&owner)
66 }
67
68 #[ink(message)]
69 pub fn approve(&mut self, spender: AccountId, value: Balance)
70     // Record the new allowance.
71     let owner = self.env().caller();
72     self.allowances.insert((owner, spender), value);
73
74     // Notify offchain users of the approval and report success.
75     self.env().emit_event(Approval {
76         owner,
77         spender,
78         value,
79     });
80     true
81 }
82
83 #[ink(message)]
84 pub fn allowance(&self, owner: AccountId, spender: AccountId)
85     self.allowance_of_or_zero(&owner, &spender)
86 }
87
88 #[ink(message)]
89 pub fn transfer_from(&mut self, from: AccountId, to: AccountId)
90     // Ensure that a sufficient allowance exists.
91     let caller = self.env().caller();
92     let allowance = self.allowance_of_or_zero(&from, &caller)
93     if allowance < value {
94         return false;
95     }
96
97     // Decrease the value of the allowance and transfer the tokens.
98     self.allowances.insert((from, caller), allowance - value)
99     self.transfer_from_to(from, to, value)
```

```
100     }
101
102     #[ink(message)]
103     pub fn transfer(&mut self, to: AccountId, value: Balance) -> I
104         self.transfer_from_to(self.env().caller(), to, value)
105     }
106
107     fn transfer_from_to(&mut self, from: AccountId, to: AccountId)
108         let from_balance = self.balance_of_or_zero(&from);
109         if from_balance < value {
110             return false
111         }
112
113         // Update the sender's balance.
114         self.balances.insert(from, from_balance - value);
115
116         // Update the receiver's balance.
```

✓Potential Solution

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract(version = "0.1.0")]
6  mod erc20 {
7      use ink_core::storage;
8
9      #[ink(storage)]
10     struct Erc20 {
11         /// The total supply.
12         total_supply: storage::Value<Balance>,
13         /// The balance of each user.
14         balances: storage::HashMap<AccountId, Balance>,
15         /// Approval spender on behalf of the message's sender.
16         allowances: storage::HashMap<(AccountId, AccountId), Balance>
17     }
18
19     #[ink(event)]
20     struct Transfer {
21         #[ink(topic)]
22         from: Option<AccountId>,
23         #[ink(topic)]
24         to: Option<AccountId>,
25         #[ink(topic)]
26         value: Balance,
27     }
28
29     #[ink(event)]
30     struct Approval {
31         #[ink(topic)]
32         owner: AccountId,
33         #[ink(topic)]
34         spender: AccountId,
35         #[ink(topic)]
36         value: Balance,
37     }
38
39     impl Erc20 {
40         #[ink(constructor)]
41         fn new(&mut self, initial_supply: Balance) {
42             let caller = self.env().caller();
43             self.total_supply.set(initial_supply);
44             self.balances.insert(caller, initial_supply);
45             self.env().emit_event(Transfer {
46                 from: None,
47                 to: Some(caller),
```

```
48             value: initial_supply,
49         });
50     }
51
52     #[ink(message)]
53     fn total_supply(&self) -> Balance {
54         *self.total_supply
55     }
56
57     #[ink(message)]
58     fn balance_of(&self, owner: AccountId) -> Balance {
59         self.balance_of_or_zero(&owner)
60     }
61
62     #[ink(message)]
63     fn approve(&mut self, spender: AccountId, value: Balance) -> bool {
64         let owner = self.env().caller();
65         self.allowances.insert((owner, spender), value);
66         self.env().emit_event(Approval {
67             owner,
68             spender,
69             value,
70         });
71         true
72     }
73
74     #[ink(message)]
75     fn allowance(&self, owner: AccountId, spender: AccountId) -> Balance {
76         self.allowance_of_or_zero(&owner, &spender)
77     }
78
79     #[ink(message)]
80     fn transfer_from(&mut self, from: AccountId, to: AccountId, value: Balance) -> bool {
81         let caller = self.env().caller();
82         let allowance = self.allowance_of_or_zero(&from, &caller);
83         if allowance < value {
84             return false
85         }
86         self.allowances.insert((from, caller), allowance - value);
87         self.transfer_from_to(from, to, value)
88     }
89
90     #[ink(message)]
91     fn transfer(&mut self, to: AccountId, value: Balance) -> bool {
92         let from = self.env().caller();
93         self.transfer_from_to(from, to, value)
94     }
95
96     fn transfer_from_to(&mut self, from: AccountId, to: AccountId, value: Balance) -> bool {
97         let from_balance = self.balance_of_or_zero(&from);
98         if from_balance < value {
99             return false
100        }
```

```
100         }
101         let to_balance = self.balance_of_or_zero(&to);
102         self.balances.insert(from, from_balance - value);
103         self.balances.insert(to, to_balance + value);
104         self.env().emit_event(Transfer {
105             from: Some(from),
106             to: Some(to),
107             value,
108         });
109         true
110     }
111
112     fn balance_of_or_zero(&self, owner: &AccountId) -> Balance {
113         *self.balances.get(owner).unwrap_or(&0)
114     }
```

Testing our Contract

- Walk through the UI and test out everything including events.

WASM Advanced

Here we will proceed to more advanced tutorials. We are preparing content. □

Meantime you can [have look on ready-made examples](#)

Fast track

If you want to play, we compiled WASM and generated metadata for you

- erc20 [WASM METADATA](#)
- erc721 [WASM METADATA](#)
- flipper [WASM METADATA](#)
- trait-erc20 [WASM METADATA](#)
- trait-flipper [WASM METADATA](#)
- multisig_plain [WASM METADATA](#)
- incrementer [WASM METADATA](#)
- delegator [WASM METADATA](#)
- dns [WASM METADATA](#)
- contract-terminate [WASM METADATA](#)
- ~~contracttransfer~~

In case some links are broken

EVM Basics

Introduction

This chapter will get you started developing smart contracts with Edgeware EVM.

We will build and deploy a simple `ERC-20` contract.

Over the course of this chapter you will learn:

- Setting up a Edgeware EVM node
- Connect Metamask to Edgeware EVM
- Using Remix with Edgeware EVM
- Using Truffle with Edgeware EVM
- Using web3.js/ethers.js with Edgeware EVM

Setting up an Edgeware EVM node

Setting up a Edgeware Node for Ethereum/EVM development

This guide walks you through setting up an Edgeware node with Ethereum/EVM compatibility.

Note This is a fast-track way to run a node. You can always compile from source as well. We recommend using your own compiled binaries for production mainnet.

Note If you don't have Docker installed, you can quickly install it from [here](#)

You can clone our repo with docker-compose to get started right away:

```
1 git clone https://github.com/hicommonwealth/edgeware-node; cd edgeware-node  
2 docker-compose up
```

Note If you want to reset or purge the local chain, delete the docker container by running `docker-compose rm`

You will see something like this:



Running-Edgeware-EVM-node

Afterwards you can head to [Polkadot Apps](#) and connect to 127.0.0.1:9944, and you should see blocks being produced.



Edgeware-EVM-producing-blocks

Now you can continue to connect [Metamask](#), Remix, and Web3.js to have great experience.

Reach us for more engagement

Glad you've made it through! □ We are eager to guide your more on your exploration through Edgeware Ethereum compatibility feature. We are **keen to hear your experience and suggestions you may have for us..** You can feel free to [chat with us in the Edgeware's channels like Discord, Element and Telegram](#), we can help you out with issues you may have or project you may want to be funded through our [Treasury program](#). Don't hesitate to share your feedback on our channels, there is always space to improve! □

Using Metamask

Introduction

This guide will show you steps for using a self-contained Edgeware dev node to send tokens between EVM accounts with Metamask. To setup your own local node, learn more at this tutorial.

In this tutorial we will use the web3 rpc endpoints to interact with Edgeware

Install the Metamask Extension

First, we start with a fresh and default [Metamask installation from the Chrome store](#). Follow the "Get Started" guide, where you need to create a wallet, set a password and store your secret backup phrase. (this gives direct access to your funds, so make sure to store these in a secure place).

Import Developer Account

Once completed, we will import our dev account. Click on upper right corner for accounts and hit Import Account :



We have prefunded developer account for this purpose:

Address: 0x19e7e376e7c213b7e7e7e46cc70a5dd086daff2a

On the import screen, select "**Private Key**" and paste there private key listed above and hit **Import**:



You should see that account imported with wild balance (123456.123E) for our needs, in our case it's **Account 4**, it may differ in your environment.



Connect to the Local Edgeware Developer Node

Now let's connect Metamask to our locally running Edgeware EVM node. On upper right, hit Networks and click Custom RPC



Metamask-Custom-RPC

Put there credentials Network Name: **Edgeware EVM** New RPC URL:

`http://127.0.0.1:9933` ChainID: `2021`

and hit **Save** button. You can see it in figure below

Make a Transfer

Now to verify your setup, you can try to make transfer between accounts. Don't worry, it's free! ;)

As new account, you should notice Nonce should be 0

Once is transaction in the block, you should see confirmed transaction like this



Reach us for more engagement

Glad you've made it through! ☐ We are eager to guide your more on your exploration through Edgeware Ethereum compatibility feature. We are **keen to hear your experience and suggestions you may have for us..** You can feel free to [chat with us in the Edgeware's channels like Discord, Element and Telegram](#), we can help you out with issues you may have or project you may want to be funded through our [Treasury program](#). Don't hesitate to share your feedback on our channels, there is always space to improve! ☐

Using Remix - Ethereum IDE

Introduction

This guide walks through the process of creating and deploying a Solidity-based smart contract to a Edgeware dev node using the [Remix IDE](#). Remix is one of the commonly used development environments for smart contracts on Ethereum. Given Edgeware's Ethereum compatibility features, Remix can be used directly with a Edgeware node.

This guide assumes that you have a running local Edgeware node running in `--dev` mode, and that you have a MetaMask installation configured to use this local node. [You can find instructions for running a local Edgeware EVM node](#) and to [configure MetaMask for Edgeware](#).

Interacting With Edgeware Using Remix

Open [Remix](#) and click on the `New File`

Name your file, in our case we've named it `erc20.sol` - yes, that [famous token standard](#)



Now add code. Here we are using simple ERC-20 contract based on the current Open Zeppelin ERC-20 template. It creates `MyFirstToken` with symbol `HEDGE` and mints the entirety of the initial supply to the creator of the contract.

```
1 pragma solidity ^0.6.0;
2
3 import 'https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-1.11.0/contracts/token/ERC20/ERC20.sol';
4
5 // This ERC-20 contract mints the specified amount of tokens to the contract's
6 contract MyFirstToken is ERC20 {
7     constructor(uint256 initialSupply) ERC20("MyFirstToken", "HEDGE") public {
8         _mint(msg.sender, initialSupply);
9     }
10 }
```



On the left sidebar, you will click on Solidity compiler and Compile erc20.sol

Solidity Compiler >>> Compile Contract



Now click to the Deploy & Run Transactions on the left in the sidebar and open Metamask to check if is our account connected. If it's connected you can skip to next steps



Select our account, in this case it's Account 4 and click on connect



You will now head to deploy contract. Just before that, make sure you've set set ENVIRONMENT to Injected Web3 and Account that we've imported. Hint, it should have some Eth. To the input next deploy input initialSupply , in our case it's 21M. Since this contract uses the default of 18 decimals, the value you will put there is

2100000000000000000000000000000000



You will hit confirm!



Remix-MM-confirm

You will see your contract has been successfully deployed.



Remix-deployed-contract

You can see your contract deployment details, that has been successfully deployed on Edgeware EVM



You can now click to call functions like `decimals`, `name`, `symbol`, `totalSupply`



What's next?

You can copy your contracts address and add it to Metamask to play out! Have fun, stay safe!

Reach us for more engagement

Glad you've made it through! ☺ We are eager to guide you more on your exploration through Edgeware Ethereum combability feature. We are **keen to hear your experience and suggestions you may have for us..** You can feel free to [chat with us in the Edgeware's channels like Discord, Element and Telegram](#), we can help you out with issues you may have or project you may want to be funded through our [Treasury program](#). Don't hesitate to share your feedback on our channels, there is always space to improve! ☺

Using Truffle

Interacting with Edgeware EVM using Truffle

Introduction

This guide walks through the process of deploying a Solidity-based smart contract to a Edgeware node using [Truffle](#). Truffle is one of the commonly used development tools for smart contracts on Ethereum. Given Edgeware's Ethereum compatibility features, Truffle can be used directly with a Edgeware node.

This guide assumes that you have a [running local Edgeware EVM node running in --dev mode..](#)

Environment Prerequisites

Installed **Nodejs** and particular package manager like **yarn** or **npm**, rest we have batteries included in this tutorial. When you are ready, clone our tutorial repository with prepared stack for you

```
git clone https://github.com/edgeware-builders/tutorials tutorials;cd tutorials
```

It will move to your cloned repository, install required packages and you are ready to go!

Let's take sneak peak to `truffle-config.js` in `truffle/` directory

```
10    networks: {
11      development: {
12        provider: () => new HDWalletProvider({
13          privateKeys: [ privKey ],
14          providerOrUrl: "http://localhost:9933/",
15        }),
16        network_id: 2021,
17      },
18    }
19 }
```

You notice few facts from here, our chainId is `2021` and we are using solc version above `^0.6.0`.

Note We are using the same private key that we have been using in other guides, which comes pre-funded with tokens `tEDG` via the genesis config of a Edgeware EVM node running in `--dev` mode. The public key for this account is:

`0x19e7e376e7c213b7e7e7e46cc70a5dd086daff2a`.

The contract we will be deploying with Truffle is a simple ERC-20 contract. You can find this contract under `truffle/contracts/HedgeToken.sol`, it's content is showed here

ERC-20 Contract

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.6.0;
3
4 import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
5
6 contract HedgeToken is ERC20 {
7   constructor(uint256 initialSupply) public ERC20("HedgeToken", "HEDGE")
8     _mint(msg.sender, initialSupply);
9   }
10 }
```

It's Simple ERC-20 contracted based on the OpenZepplin ERC20 contract that creates `HedgeToken` and assigns the created initial token supply to the contract creator.

You can notice initial supply in `migrations/2_deploy.contracts.js`, it contains the following:

```
1 var HedgeToken = artifacts.require("HedgeToken");
2
3 module.exports = function (deployer) {
4   deployer.deploy(HedgeToken, "21000000000000000000000000000000");
5 };
```

`21000000000000000000000000000000` is the number of tokens to initially mint with the contract, that is 21 Million with 18 decimal places. [Since OpenZepplin v3.0+, there is default decimal 18 for SimpleToken.sol](#)

Now let's go to the essential part! After you had installed necessary packages, continue in terminal with following command

Compile ERC-20 Contract

```
npx truffle compile
```



What id does, it take OpenZepplin ERC20.sol token, compiles it with other referenced code in other OpenZepplin code, creates artifact (bytecode) and ABI (contract interface)

Deploying a Contract to Edgeware EVM Using Truffle

Now let's go to the hot stuff, deploy it to our Edgeware EVM

```
npx truffle --network development migrate
```



As you may see, we are using our `development` network from `truffle-config.js`. From migrate you'll notice there what our contract address is of our contract.

Reach us for more engagement

Glad you've made it through! □ We are eager to guide you more on your exploration through Edgeware Ethereum combability feature. We are **keen to hear your experience and suggestions you may have for us..** You can feel free to [chat with us in the Edgeware's channels like Discord, Element and Telegram](#), we can help you out with issues you may have or project you may want to be funded through our [Treasury program](#). Don't hesitate to share your feedback on our channels, there is always space to improve! □

Using Web3

Introduction

This guide walks through the process of using Web3 to manually sign and send a transaction to a Edgeware EVM dev node. For this example, we will use Node.js and straightforward JavaScript code.



Note This tutorial was created using the release of Edgeware EVM. The Edgeware EVM platform, and the [Frontier](#) components it relies on for Substrate-based Ethereum compatibility, are still under very active development. We have created this tutorial so you can test out Edgeware's Ethereum compatibility features. Even though we are still in development, we believe it's important that interested community members and developers have the opportunity to start to try things with Edgeware and provide feedback.

Checking Prerequisites

Installed [Nodejs](#) and particular package manager like [yarn](#) or [npm](#), rest we have batteries included in this tutorial. This guide assumes that you have a [running local Edgeware EVM node running in --dev mode..](#)

```
git clone https://github.com/edgeware-builders/tutorials tutorials;cd tutorials
```

It will move to your cloned repository, install required packages and you are ready to go!

Creating Transaction

For this example, we only need a single JavaScript file to create the transaction, which we will run using the `node` command in the terminal. The script will transfer 1337 ETH from the genesis account to another address. For simplicity, the file is divided into three sections: variable definition, create transactions and broadcast transaction.

We need to set a couple of values in the variables definitions:

- Create our Web3 constructor (`web3`)
 - Define the `privKey` variable as the private key of our genesis account, which is where all the fund are stored when deploying your local Edgeware EVM node and what is used to sign the transactions
 - Set the "from" and "to" address, making sure to set the value of `toAddress` to a different address, for example the one created by Metamask when setting up a local wallet

Both the *create transaction* and *deploy transaction* sections are wrapped in an asynchronous function that handles the promises from our Web3 instance. To create the transaction, we use the `web3.eth.accounts.signTransaction(tx, privKey)` command, where we have to define the tx object with some parameters such as: `addressFrom` , `addressTo` , `number of tokens to send` , and the `gas limit` .

```
1 const deploy = async () => {
2   console.log(
3     `Attempting to make transaction from ${addressFrom} to ${addressTo}`),
4   );
5
6   const createTransaction = await web3.eth.accounts.signTransaction(
7     {
8       from: addressFrom,
9       to: addressTo,
10      value: web3.utils.toWei('1337', 'ether'),
11      gas: 21000,
12    },
13    privKey
14  );

```

Complete `createTransaction.js` should look like this!

Check balance on accounts

Before running the script, we need another file to check the balances of both addresses before and after the transaction is executed. We can easily do this by leveraging the Ethereum compatibility features of Edgeware.

`balances.js` looks like this:

```
1 const Web3 = require('web3');
2
3 // Variables definition
4 const addressFrom = '0x19e7e376e7c213b7e7e46cc70a5dd086daff2a';
5 const addressTo = '0x6bB5423f0Dd01B8C5028a1bc01e1f1bDe4523e72';
6 const web3 = new Web3('http://127.0.0.1:9933');
7
8 // Balance call
9 const balances = async () => {
10     const balanceFrom = web3.utils.fromWei(
11         await web3.eth.getBalance(addressFrom),
12         'ether'
13     );
14     const balanceTo = await web3.utils.fromWei(
15         await web3.eth.getBalance(addressTo),
16         'ether'
17     );
18
19     console.log(`The balance of ${addressFrom} is: ${balanceFrom} ETH.`);
20     console.log(`The balance of ${addressTo} is: ${balanceTo} ETH.`);
21 };
22
23 balances();Copy to clipboardErrorCopied
```

Play time

Run `node balance.js` to check initial balance on accounts



Run `node createTransaction.js` to transfer some stuff around chain



Run `node balance.js` to check result balances



Reach us for more engagement

Glad you've made it through! ☐ We are eager to guide you more on your exploration through Edgeware Ethereum combability feature. We are **keen to hear your experience and suggestions you may have for us..** You can feel free to [chat with us in the Edgeware's channels like Discord, Element and Telegram](#), we can help you out with issues you may have or project you may want to be funded through our [Treasury program](#). Don't hesitate to share your feedback on our channels, there is always space to improve! ☐

Using Hardhat



This page is still under development. Many functionalities of this page are still being worked on until further notice. As stability is updated, add a pull request with revised changes, thank you.



Interacting with Edgeware EVM using Hardhat

This guide walks through the process of deploying a **Solidity-based smart contract** to the Edgeware testnet Beresheet using Hardhat. **Hardhat** is a development environment to compile, deploy, test, and debug your Ethereum software. It helps developers manage and automate the recurring tasks that are inherent to the process of building smart contracts and dApps, as well as easily introducing more functionality around this workflow. This means compiling, running and testing smart contracts at the very core.

Hardhat comes built-in with **Hardhat Network**, a local Ethereum network designed for development. Its functionality focuses around Solidity debugging, featuring stack traces, `console.log()` and explicit error messages when transactions fail.

Hardhat Runner, the CLI command to interact with Hardhat, is an extensible task runner. It's designed around the concepts of tasks and plugins. Every time you're running Hardhat from the CLI you're running a task. E.g. `npx hardhat compile` is running the built-in compile task. Tasks can call other tasks, allowing complex workflows to be defined. Users and plugins can override existing tasks, making those workflows customizable and extendable.

A lot of Hardhat's functionality comes from plugins, and, as a developer, you're free to choose which ones you want to use. Hardhat is unopinionated in terms of what tools you end up using, but it does come with some built-in defaults. All of which can be overridden.

To follow this tutorial you should be able to:

- Write code in JavaScript
- Operate a terminal
- Use git
- Understand the basics of how smart contracts work
- Set up a Metamask wallet

Installation

We need to install Node.js and npm package manager. You can download directly from Node.js or in your terminal.

Mac OS

```
1 # You can use homebrew (https://docs.brew.sh/Installation)
2 brew install node`  
3
4 # Or you can use nvm (https://github.com/nvm-sh/nvm)
5 nvm install node
```

```
1 curl -sL https://deb.nodesource.com/setup_15.x | sudo -E bash -
2
3 sudo apt install -y nodejs
```

You can verify that everything is installed correctly by querying the version for each package:

```
node -v
```

```
npm -v
```

As of writing of this guide, the versions used were 15.7.0 and 7.4.3, respectively.

Also, you will need the following:

- Have MetaMask installed and connected to Beresheet
- Have an account with funds, which you can get from the automated bot on the Edgeware discord (*in construction*)

Once all requirements have been met, you are ready to build with Hardhat.

Starting a Hardhat Project

To start a new project, create a directory for it:

```
mkdir EDGhat && cd EDGhat
```

Then, initialize the project by running:

```
npm init -y
```

You will notice a newly created `package.json`, which will continue to grow as you install project dependencies.

To get started with Hardhat, we will install it in our newly created project directory:

```
npm install hardhat
```

Once installed, run:

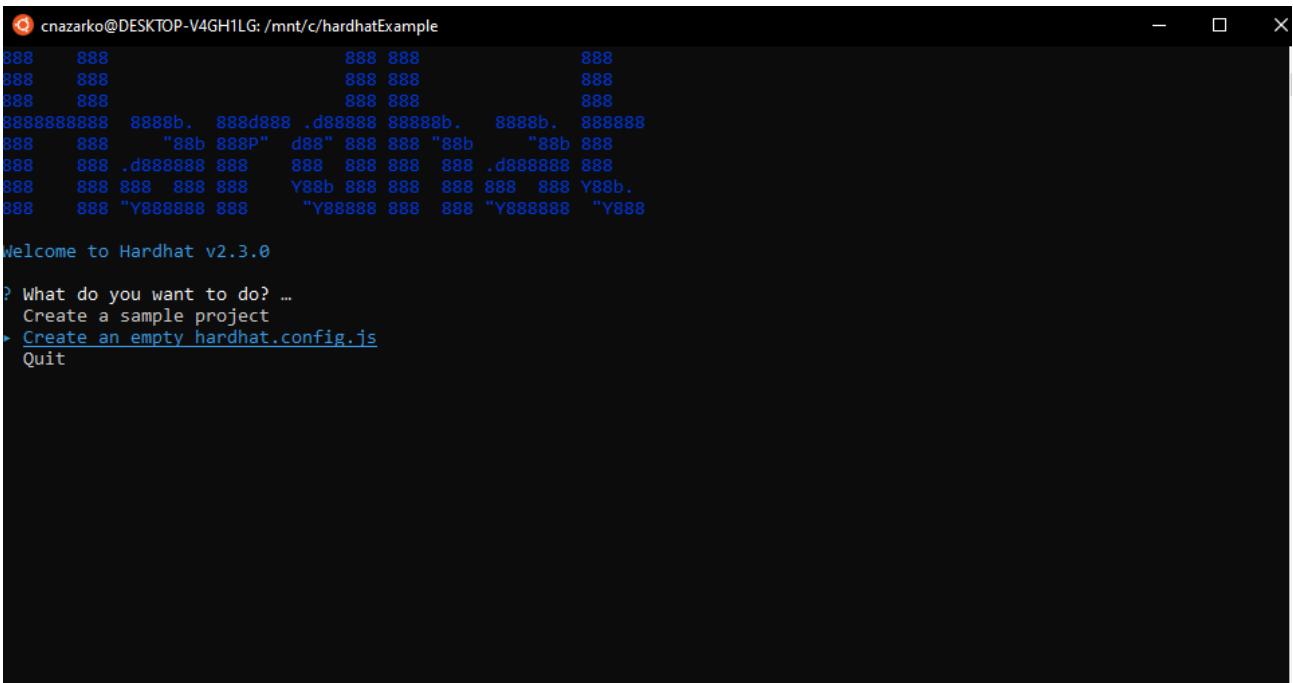
```
npx hardhat
```

This will create a Hardhat config file (`hardhat.config.js`) in our project directory.



`npx` is used to run executables installed locally in your project. Although Hardhat can be installed globally, we recommend installing locally in each project so that you can control the version on a project by project basis.

After running the command, choose `Create an empty hardhat.config.js` by using the arrow keys and enter:



```
cnazarko@DESKTOP-V4GH1LG: /mnt/c/hardhatExample
888     888          888 888          888
888     888          888 888          888
888     888          888 888          888
8888888888 8888b. 888d888 .d88888 88888b. 8888b. 888888
888     888 "88b 888P" d88" 888 888 "88b      "88b 888
888     888 .d888888 888    888 888 888 888 .d888888 888
888     888 888 888    Y88b 888 888 888 888 888 Y88b.
888     888 "Y888888 888    "Y888888 888 "Y8888888 "Y888

Welcome to Hardhat v2.3.0

? What do you want to do? ...
Create a sample project
• Create an empty hardhat.config.js
Quit
```

The contract file

We are going to store our contract in the `contracts` directory. Create it:

```
mkdir contracts && cd contracts or by creating a new directory under our root directory  
'EDGHAT' in a text-editor such as Visual Studio Code
```

(i) Editing directories in the terminal can be replicated in a text-editor at any point in this tutorial.

The smart contract that we'll deploy as an example will be called Box: it will let people store a value that can be later retrieved.

We will save this file as `contracts/Box.sol`:

```
1 // contracts/Box.sol
2 pragma solidity ^0.8.1;
3
4 contract Box {
```

```
5     uint256 private value;
6
7     // Emitted when the stored value changes
8     event ValueChanged(uint256 newValue);
9
10    // Stores a new value in the contract
11    function store(uint256 newValue) public {
12        value = newValue;
13        emit ValueChanged(newValue);
14    }
15
16    // Reads the last stored value
17    function retrieve() public view returns (uint256) {
18        return value;
19    }
20 }
```

Hardhat Configuration File

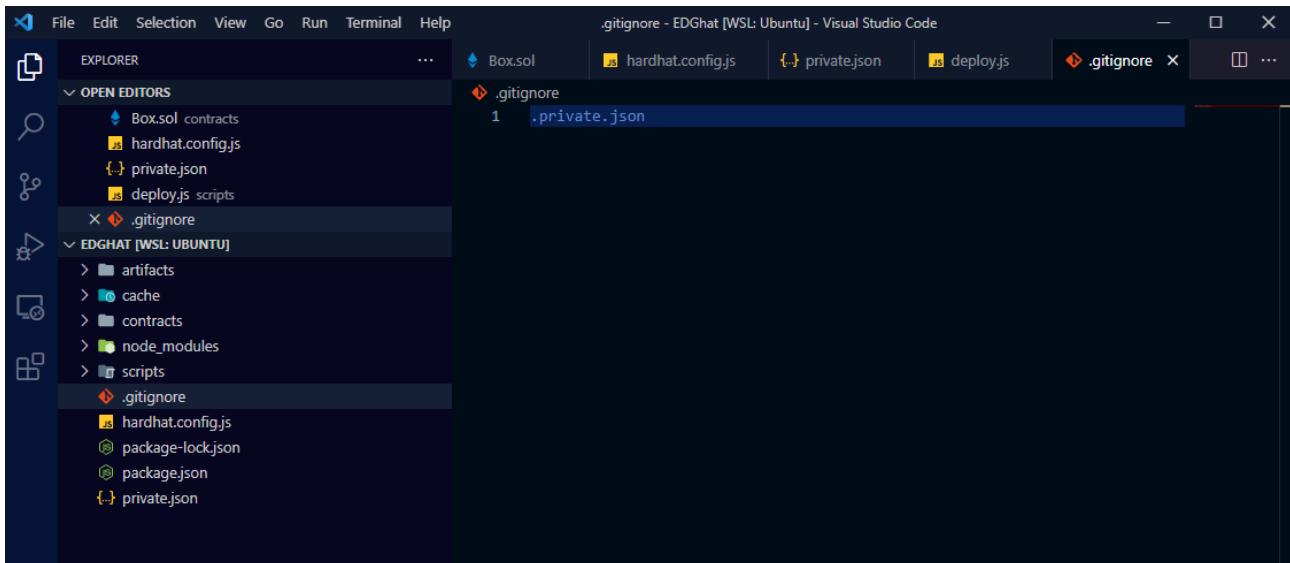
We need to modify our Hardhat configuration file so we can compile and deploy contracts into the Edgeware ecosystem. If you have not yet done so, connect your MetaMask Account to our ecosystem and fund it with the automated faucet bot on discord. We will use the private key of the account created to deploy the contract. If you don't know the private key, you can export it from MetaMask.

We start by requiring the [ethers plugin](#), which brings the [ethers.js]/[integrations/ethers/] library that allows you to interact with the blockchain in a simple way. We can install ethers plugin by running:

```
npm install @nomiclabs/hardhat-ethers ethers
```

Next, we import the private key that we've retrieved from MetaMask and store it in a `.json` file. This file should be created under the root directory, and named `private.json`. Because this key is highly sensitive information, it's very important that we are not revealing any information when deploying. To ensure this, we want to create a `.gitignore` file under our root directory. Then, you can ignore any files by using the format: `.filename` or any

directories by using: `directory/ .` In our case, we are trying to ignore our private key file so it should look like this:



The `private.json` file must contain a `privateKey` entry, for example:

```
1  {
2    "privateKey": "YOUR-PRIVATE-KEY-HERE"
3 }
```

Inside the `module.exports`, we need to provide the Solidity version (`0.8.1` according to our contract file), and the network details. Here, we are using testnet(Beresheet) network for the following example :

- Network name: Beresheet
- RPC URL: <https://beresheet2.edgewa.re/evm> (Alternatively, one can use <https://beresheetX.edgewa.re/evm> where X can be any number from 1 to 8.)
- Chain ID: 2021
- gas: 180000

If you want to deploy to a local Edgeware development node, you can use the following network details:

- Network name: dev
- RPC URL: <http://localhost:9933/>

- Chain ID: 2021
- gas: 180000

If you want to deploy on the Edgeware mainnet, you can use the following network details:

- Network name: Edgeware
- RPC URL: <https://mainnet2.edgewa.re/evm> (Alternatively, one can use <https://mainnetX.edgewa.re/evm> where X can be any number from 1 to 20.)
- Chain ID: 2021
- gas: 180000

The Hardhat configuration file should look like this:

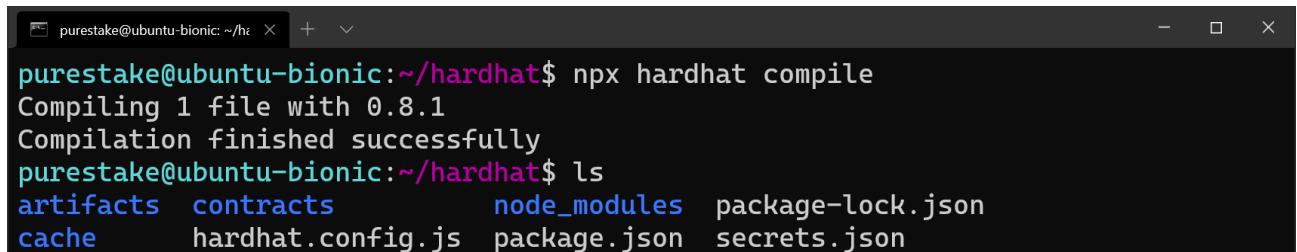
```
1 // ethers plugin required to interact with the contract
2 require('@nomiclabs/hardhat-ethers');
3
4 // private key from the pre-funded Beresheet testing account
5 const { privateKey } = require('./private.json');
6
7 module.exports = {
8   // latest Solidity version
9   solidity: "0.8.1",
10
11   networks: {
12     // Beresheet network specification
13     Beresheet: {
14       url: `https://beresheet2.edgewa.re/evm`,
15       chainId: 2021,
16       gas: 180000,
17       accounts: [privateKey]
18     }
19   }
20};
```

Great! We are now ready for deployment.

Compiling Solidity

Our contract, `Box.sol`, uses Solidity 0.8.1. Make sure the Hardhat configuration file is correctly set up with this solidity version. If so, we can compile the contract by running:

```
npx hardhat compile
```



```
purestake@ubuntu-bionic:~/hardhat$ npx hardhat compile
Compiling 1 file with 0.8.1
Compilation finished successfully
purestake@ubuntu-bionic:~/hardhat$ ls
artifacts  contracts      node_modules  package-lock.json
cache       hardhat.config.js  package.json  secrets.json
```

A terminal window titled "purestake@ubuntu-bionic: ~/hardhat" shows the command "npx hardhat compile" being run. The output indicates that 1 file is being compiled with Solidity 0.8.1 and the compilation is successful. The directory listing shows files like artifacts, contracts, node_modules, package-lock.json, cache, hardhat.config.js, package.json, and secrets.json.

After compilation, an `artifacts` directory is created: it holds the bytecode and metadata of the contract, which are `.json` files. It's a good idea to add this directory to your `.gitignore`.

Deploying the Contract

In order to deploy the Box smart contract, we will need to write a simple `deployment script`. First, let's create a new directory (`scripts`). Inside the newly created directory, add a new file `deploy.js`.

```
1 mkdir scripts && cd scripts
2 touch deploy.js
```

Next, we need to write our deployment script using `ethers`. Because we'll be running it with Hardhat, we don't need to import any libraries.

We start by creating a local instance of the contract with the `getContractFactory()` method. Next, let's use the `deploy()` method that exists within this instance to initiate the smart contract. Lastly, we wait for its deployment by using `deployed()`. Once deployed, we can fetch the address of the contract inside the box instantiation.

```
1 // scripts/deploy.js
2 async function main() {
3     // We get the contract to deploy
4     const Box = await ethers.getContractFactory('Box');
5     console.log('Deploying Box...');
6
7     // Instantiating a new Box smart contract
8     const box = await Box.deploy();
9
10    // Waiting for the deployment to resolve
11    await box.deployed();
12    console.log('Box deployed to:', box.address);
13 }
14
15 main()
16 .then(() => process.exit(0))
17 .catch((error) => {
18     console.error(error);
19     process.exit(1);
20});
```

Using the run command, we can now deploy the Box contract to Beresheet.

```
npx hardhat run --network Beresheet scripts/deploy.js
```

 To deploy to a Edgeware development node, replace Beresheet for dev in the run command.

After a few seconds, the contract is deployed, and you should see the address in the terminal.

```
cnazarko@DESKTOP-V4GH1LG:/mnt/c/EDGhat
cnazarko@DESKTOP-V4GH1LG:/mnt/c/EDGhat$ npx hardhat run --network Beresheet scripts/deploy.js
Deploying Box...
Box deployed to: 0x470709a40F17aD2FFFcA1504481b8FD32b042080
```

Congratulations, your contract is live! Save the address, as we will use it to interact with this contract instance in the next step.

Interacting with the Contract

 Limited functionality

Let's use Hardhat to interact with our newly deployed contract in Beresheet. To do so, launch hardhat console by running:

```
npx hardhat console --network Beresheet
```

 To deploy to a Edgeware development node, replace Beresheet for dev in the run command.

Then, add the following lines of code one line at a time. First, we create a local instance of the `Box.sol` contract once again. Don't worry about the `undefined` output you will get after each line is executed:

```
const Box = await ethers.getContractFactory('Box');
```

Next, let's connect this instance to an existing one by passing in the address we obtained when deploying the contract:

```
const box = await Box.attach('ADDRESS-GOES-HERE');
```

After attaching to the contract, we are ready to interact with it. While the console is still in session, let's call the store method and store a simple value:

```
await box.store(5)
```

The transaction will be signed by your Beresheet account and broadcast to the network. The output should look similar to:

Notice your address labeled from, the address of the contract, and the data that is being passed. Now, let's retrieve the value by running:

```
(await box.retrieve()).toNumber()
```

We should see 5 or the value you have stored initially.

Congratulations, you have completed the Hardhat tutorial!

For more information on Hardhat, hardhat plugins, and other exciting functionality, please visit hardhat.org.

EVM Resources

- Set of various web3 utilities/approaches to give you technical overview
- Convert Substrate address to Ethereum address - Convert Address

WASM Ballot Contract

Introduction

In this chapter, we will teach you how to use ink! to write more complex contracts.

We will build a "Ballot" contract which will allow users to add proposals to a ballot and vote on them. The users will be able to register themselves as a voter with a function call and there will be a chair person (owner) of the contract that will oversee the process.

Over the course of this chapter you will learn:

- To build custom structs
- To store custom structs in vectors and hash maps
- To safely get and update the structs in these collections
- To use ink_prelude crate
- To use traits like Clone, Debug, PackedLayout and SpreadLayout

Contract Template

Let's start with making a new ink! project to build the ballot contract.

In your working directory, run:

```
cargo contract new ballot
```

Replace the content of `lib.rs` file with the template on the right.

The contract storage consists of an `AccountId` which we initialize to the callers id in the constructor. There is a function `get_chair_person` implemented that returns the id of the chair_person(owner) of the contract.

Struct

You may have come across the `struct` keyword in previous tutorials, but so far we have used structs to define the storage of contracts. In this contract, we use it to define the following custom types that are going to be used later use as part ballot storage:

- `Proposal` : This struct stores information about a proposal. Each proposal contains:
 - `name` : A field to store the name of the proposal
 - `vote_count` : A 32 bit unsigned integer for storing the number of votes the proposal has received.
- `Voter` : For each voter in the system, we instantiate a voter struct with:
 - `weight` : An unsigned weight indicating the weightage of the voter. The weightage of a voter can vary based on election/network parameters.
 - `voted` : It is set to false by default, but once a voter has voted, it's set to true so that the same voter can not cast his vote again.
 - `delegate` : A voter can choose to delegate their vote to some one else. Since it's not necessary for voters to delegate, this field is created as an `option` .

- `vote` : Index of the proposal that the user has casted the vote to. This is created as an `Option` set to None by default.

Unlike our contract struct `Ballot` we don't use the macro `ink(storage)` for our custom defined structs as there can only be a single storage struct for a contract. Also, our structs are not public as users don't need to interact with them directly.

Ink_Prelude

`ink_prelude` crate provides data structures such as `HashMap`, `Vector` etc.. to operate on contract memory during contract execution. We will be importing these collections in next parts of this tutorial so before moving forward update contract's cargo.toml file with following dependency:

```
ink_prelude = { version = "3.0.0-rc2", default-features = false }
```

Compilaton and Warnings

You can build the contract using `cargo +nightly build` and run tests using `cargo +nightly test`. The contract will successfully compile and pass all tests, but the rust compiler will give you the following warnings:

```
1 warning: struct is never constructed: `Proposal`
2     --> lib.rs:10:12
3     |
4 10 |     struct Proposal {
5     |             ^^^^^^
6     |
7     = note: `#[warn(dead_code)]` on by default
8
9 warning: struct is never constructed: `Voter`
10    --> lib.rs:16:16
11    |
12 16 |     pub struct Voter {
13     |             ^^^^
14
15 warning: 2 warnings emitted
```

This is because the structs we have defined are never used. We will get to that in next part!

Starting Point

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod ballot {
7      // Structure to store Proposal information
8      struct Proposal {
9          name: String,
10         vote_count: i32,
11     }
12
13     // Structure to store Proposal information
14     pub struct Voter {
15         weight: i32,
16         voted: bool,
17         delegate: Option<AccountId>,
18         vote: Option<i32>,
19     }
20
21     /// Defines the storage of your contract.
22     /// Add new fields to the below struct in order
23     /// to add new static storage fields to your contract.
24     #[ink(storage)]
25     pub struct Ballot {
26         chair_person: AccountId,
27     }
28
29     impl Ballot {
30         #[ink(constructor)]
31         pub fn new() -> Self {
32             let owner = Self::env().caller();
33             Self {
34                 chair_person: owner,
35             }
36         }
37
38         #[ink(message)]
39         pub fn get_chairperson(&self) -> AccountId {
40             self.chair_person
41         }
42
43     }
44
45     /// Unit tests in Rust are normally defined within such a `#[cfg(
46     /// module and test functions are marked with a `#[test]` attribut
```

```
48     /// The below code is technically just normal Rust code.
49     #[cfg(test)]
50     mod tests {
51         /// Imports all the definitions from the outer scope so we can
52         use super::*;

53
54         // Alias `ink_lang` so we can use `ink::test`.
55         use ink_lang as ink;
56
57         #[ink::test]
58         fn new_works() {
59             let ballot = Ballot::new();
60             assert_eq!(ballot.get_chairperson(), AccountId::from([0x1;
61             }
62         }
63     }
```

Collection and Traits

In this part, we will be using the structs built previously and write public functions to add and fetch data from our contract.

Collections

For this contract, we are going to store our voters in a `HashMap` with `AccountId` as a key and `Voter` instance as value. The `HashMap` can be imported from the `ink_storage` crate by:

```
use ink_storage::collections::HashMap;
```

The proposals will be stored in a `Vec` collection that can be imported from the `ink_prelude` crate. `ink_prelude` is a collection of data structures that operate on contract memory during contract execution. The vector can be imported by:

```
use ink_prelude::vec::Vec;
```

Vectors can be instantiated in the same way as a `HashMap`. New objects can be added or referenced to from a vector using:

```
1  let proposals: Vec<Proposals> = Vec::new();
2  // adding a new proposal
3  proposals.push(
4      Proposal{
5          name:String::from("Proposal # 1"),
6          vote_count: 0,
7      });
8  // returns the proposal at index 0 if exists else returns None
9  let proposal = self.proposals.get(0).unwrap();
```

Remember that the `vector.get` returns an `Option` not the actual object!

Traits

A trait tells the Rust compiler about the functionality a particular type has, and can be shared with other types. You can read more about them [here](#). Before using the custom built structures inside the `Ballot` storage, certain traits are required to be implemented for `Voter` and `Proposal` structs. These traits include:

- `Debug` : Allows debug formatting in format strings
- `Clone` : This trait allows you to create a deep copy of object
- `Copy` : The copy traits allows you to copy a value of a field
- `PackedLayout` : Types that can be stored to and loaded from a single contract storage cell
- `SpreadLayout` : Types that can be stored to and loaded from the contract storage.

You can learn more about these traits over [here](#) and [here](#). These traits are implemented using the `derive` attribute:

```
1 #[derive(Clone, Copy, Debug, scale::Encode, scale::Decode, SpreadLayout
2 struct XYZ {
3     ...
4     ...
5 }
```

Your Turn!

You need to:

- Create a proposal Vec and voters HashMap in Ballot struct.
- Update the constructor, so that it initializes a Vec of proposals and a HashMap of voters. Also update the voters HashMap to include the chair person as a voter.

- Create getters for both storage items.
- Write `add_voter` function to create a voter by the given `AccountId` and insert it in the `HashMap` of voters.
- Write `add_proposal` function that creates a `Proposal` object and inserts it tot he vector of proposals.

Remember to run `cargo +nightly test` to test your work.

Starting Point

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod ballot {
7      use ink_storage::collections::HashMap;
8      use ink_prelude::vec::Vec;
9      use ink_storage::traits::{PackedLayout, SpreadLayout};
10
11     // Structure to store Proposal information
12     #[derive(Clone, Debug, scale::Encode, scale::Decode, SpreadLayout)]
13     struct Proposal {
14         name: String,
15         vote_count: u32,
16     }
17
18     // Structure to store Voter information
19     #[derive(Clone, Debug, scale::Encode, scale::Decode, SpreadLayout)]
20     pub struct Voter {
21         weight: u32,
22         voted: bool,
23         delegate: Option<AccountId>,
24         vote: Option<i32>,
25     }
26
27     /// Defines the storage of your contract.
28     /// Add new fields to the struct in order
29     /// to add new static storage fields to your contract.
30     #[ink(storage)]
31     pub struct Ballot {
32         chair_person: AccountId,
33         // ACTION: create a voters hash map with account id as key and
34         voters:
35
36         // ACTION: create a proposals vector
37         proposals:
38     }
39
40     impl Ballot {
41         #[ink(constructor)]
42         pub fn new() -> Self {
43             // get chair person address
44             let chair_person = Self::env().caller();
45
46             // ACTION: create empty proposals and voters variables
47             //           * let proposals =
```

```

48          //           * let mut voters =
49
50          // ACTION: add chair persons voter object in voters hash
51          // HINT: Use hashmap.insert(key,value)
52
53          Self {
54              chair_person,
55              voters,
56              proposals,
57          }
58      }
59
60      // return chair person id
61      #[ink(message)]
62      pub fn get_chairperson(&self) -> AccountId {
63          self.chair_person
64      }
65
66      // return the provided voter object
67      pub fn get_voter(&self, voter_id: AccountId) -> Option<&Voter>
68      {
69
70          // return the count of voters
71          pub fn get_voter_count(&self) -> usize{
72          }
73
74
75          /// the function adds the provided voter id into possible
76          /// list of voters. By default the voter has no voting right,
77          /// the contract owner must approve the voter before he can cast
78          #[ink(message)]
79          pub fn add_voter(&mut self, voter_id: AccountId) -> bool{
80              // ACTION: check if voter already exists, if yes return false
81              //           * if not exists, create an entry in hash map
82              //           * with default weight set to 0 and voted to false
83              //           * and return true
84
85              // HINT: use hashmap.get() to get voter
86              //           and use options.some() to check if voter exists
87          }
88
89          /// given an index returns the name of the proposal at that index
90          pub fn get_proposal_name_at_index(&self, index:usize) -> &String
91      }
92
93          /// returns the number of proposals in ballot
94          pub fn get_proposal_count(&self) -> usize {
95      }
96
97
98          /// adds the given proposal name in ballot
99          pub fn add_proposal(&mut self, proposal_name: String){

```

```
100      }
101
102
103
104
105      }
106
107      /// Unit tests in Rust are normally defined within such a `#[cfg(`
108      /// module and test functions are marked with a `#[test]` attribut
109      /// The below code is technically just normal Rust code.
110      #[cfg(test)]
111      mod tests {
112          /// Imports all the definitions from the outer scope so we can
113          use super::*;

114
115          // Alias `ink_lang` so we can use `ink::test`.
116          use ink_lang as ink;

117
118          #[ink::test]
119          fn new_works() {
120              let mut proposal_names: Vec<String> = Vec::new();
121              proposal_names.push(String::from("Proposal # 1"));
122              let ballot = Ballot::new();
123              assert_eq!(ballot.get_voter_count(), 1);
124          }

125
126          #[ink::test]
127          fn adding_proposals_works() {
128              let mut ballot = Ballot::new();
129              ballot.add_proposal(String::from("Proposal #1"));
130              assert_eq!(ballot.get_proposal_count(), 1);
131          }

132
133          #[ink::test]
134          fn adding_voters_work() {
135              let mut ballot = Ballot::new();
136              let account_id = AccountId::from([0x0; 32]);
137              assert_eq!(ballot.add_voter(account_id), true);
138              assert_eq!(ballot.add_voter(account_id), false);
139          }
140      }
141 }
```

✓Potential Solution

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod ballot {
7      // use Hash
8      use ink_storage::collections::HashMap;
9      use ink_prelude::vec::Vec;
10     use ink_storage::traits::{PackedLayout, SpreadLayout};
11
12     // Structure to store Proposal information
13     #[derive(Clone, Debug, scale::Encode, scale::Decode, SpreadLayout)]
14     struct Proposal {
15         name: String,
16         vote_count: u32,
17     }
18
19     // Structure to store Voter information
20     #[derive(Clone, Debug, scale::Encode, scale::Decode, SpreadLayout)]
21     pub struct Voter {
22         weight: u32,
23         voted: bool,
24         delegate: Option<AccountId>,
25         vote: Option<i32>,
26     }
27
28     /// Defines the storage of your contract.
29     /// Add new fields to the below struct in order
30     /// to add new static storage fields to your contract.
31     #[ink(storage)]
32     pub struct Ballot {
33         chair_person: AccountId,
34         voters: HashMap<AccountId, Voter>,
35         proposals: Vec<Proposal>
36     }
37
38     impl Ballot {
39         #[ink(constructor)]
40         pub fn new() -> Self {
41             // get chair person address
42             let chair_person = Self::env().caller();
43
44             // create empty proposal and voters
45             let proposals: Vec<Proposal> = Vec::new();
46             let mut voters = HashMap::new();
47         }
48     }
49 }
```

```

48         // initialize chair person's vote
49         voters.insert(chair_person, Voter{
50             weight:1,
51             voted:false,
52             delegate: None,
53             vote: None,
54         });
55
56     Self {
57         chair_person,
58         voters,
59         proposals,
60     }
61 }
62
63
64 #[ink(message)]
65 pub fn get_chairperson(&self) -> AccountId {
66     self.chair_person
67 }
68
69
70
71 pub fn get_voter(&self, voter_id: AccountId) -> Option<&Voter> {
72     self.voters.get(&voter_id)
73 }
74
75 pub fn get_voter_count(&self) -> usize{
76     self.voters.len() as usize
77 }
78
79         /// the function adds the provided voter id into poss
80         /// list of voters. By default the voter has no voting right,
81         /// the contract owner must approve the voter before he can c
82 #[ink(message)]
83 pub fn add_voter(&mut self, voter_id: AccountId) -> bool{
84
85     let voter_opt = self.voters.get(&voter_id);
86     // the voter does not exists
87     if voter_opt.is_none() {
88         return false
89     }
90
91     self.voters.insert(voter_id, Voter{
92         weight:0,
93         voted:false,
94         delegate: None,
95         vote: None,
96     });
97     return true
98 }
99

```

```
100
101
102     /// given an index returns the name of the proposal at that index
103     pub fn get_proposal_name_at_index(&self, index:usize) -> &String {
104         let proposal = self.proposals.get(index).unwrap();
105         return &proposal.name
106     }
107
108     /// returns the number of proposals in ballot
109     pub fn get_proposal_count(&self) -> usize {
110         return self.proposals.len()
111     }
112
113     /// adds the given proposal name in ballot
114     /// to do: check uniqueness of proposal,
115     pub fn add_proposal(&mut self, proposal_name: String){
116         self.proposals.push(
117             Proposal{
118                 name:String::from(proposal_name),
119                 vote_count: 0,
120             });
121     }
122
123 }
124
125     /// Unit tests in Rust are normally defined within such a `#[cfg(test)]` block.
126     /// module and test functions are marked with a `#[test]` attribute.
127     /// The below code is technically just normal Rust code.
128 #[cfg(test)]
129 mod tests {
130     /// Imports all the definitions from the outer scope so we can use them here.
131     use super::*;

132     // Alias `ink_lang` so we can use `ink::test`.
133     use ink_lang as ink;

134     #[ink::test]
135     fn new_works() {
136         let mut proposal_names: Vec<String> = Vec::new();
137         proposal_names.push(String::from("Proposal # 1"));
138         let ballot = Ballot::new();
139         assert_eq!(ballot.get_voter_count(), 1);
140     }
141 }
```

Adding Functionality

In this part we will add functionality to our Ballot so that:

- People can vote on proposals
- People can delegate their votes
- The chairperson can assign voting rights

Contract Functionality

Constructor:

Let's first update the constructor of our contract. As you can see in the code sample on the right, the constructor now accepts a `Option<Vector<String>>` parameter. The constructor expects a vector of strings as input. We need to update the constructor so that the provided proposal names are used to create the `Proposal` objects and added to our ballot storage. To check if the vector containing strings is provided:

```
1  if proposal_name.is_some() {  
2      names = proposal_name.unwrap()  
3      // do something with names  
4  }  
5 }
```

Give Voting Right:

In the previous part we created a function that allowed users to add themselves as a voter. We initialized their voter struct with `voter.weight=0` because when a voter is created, he/she has no voting right by default. So, let's create a function `give_voting_right` that will only allow the chairperson to update the weight to 1 for any given voter. The function will look something like:

```
1  // assuming that the caller is the chair person  
2  pub fn give_voting_right(&mut self, voter_id: AccountId) {  
3      let voter_opt = self.voters.get_mut(&voter_id);
```

```
4     if voter_opt.is_some() {
5         let voter = voter.unwrap()
6         // assuming that the voter has not already voted
7         voter.weight = 1
8     }
9 }
```

Vote:

Now, let's implement a function that will allow users to cast their votes. This function will take a proposal index as input. If the `caller` is a valid voter and has not already casted his/her vote, update the proposal at index `i` with the weight of the voter, update `voter.voted` to true and set `voter.vote` to index `i`.

Get Winning Proposal:

Now that the votes are cast, we will implement a function that will get the name of the winning proposal. In a recall election, the winner is announced once the voting time has passed out. We will leave such implementation to you. For now, we will allow any user to invoke this function and get the name of the winning proposal. Let's implement a function to return the index of the proposal with the maximum votes:

```
1 fn winning_proposal(&self) -> Option<usize> {
2     let mut winning_vote_vount:u32 = 0;
3     let mut winning_index: Option<usize> = None;
4     let mut index: usize = 0;
5
6     for val in self.proposals.iter() {
7         if val.vote_count > winning_vote_vount {
8             winning_vote_vount = val.vote_count;
9             winning_index = Some(index);
10        }
11        index += 1
12    }
13    return winning_index
14 }
15 }
```

Notice that this function returns `Option<usize>`, not `usize`, since it's possible that there are no proposals in the ballot. This function can be used to find the name of winning

proposal.

Delegation:

In our voter struct, there is a `delegate` field defined as `Option<AccountId>` to allow voters to delegate their vote to someone else. This can be achieved using the following function:

```
1 #[ink(message)]
2 pub fn delegate(&mut self, to: AccountId) {
3
4     // account id of the person who invoked the function
5     let sender_id = self.env().caller();
6     let sender_weight;
7     // self delegation is not allowed
8     assert_ne!(to, sender_id, "Self-delegation is disallowed.");
9
10    {
11        let sender_opt = self.voters.get_mut(&sender_id);
12        // the voter invoking the function should exist in our ballot
13        assert_eq!(sender_opt.is_some(), true, "Caller is not a valid voter");
14        let sender = sender_opt.unwrap();
15
16        // the voter must not have already casted their vote
17        assert_eq!(sender.voted, false, "You have already voted");
18
19        sender.voted = true;
20        sender.delegate = Some(to);
21        sender_weight = sender.weight;
22    }
23
24    {
25        let delegate_opt = self.voters.get_mut(&to);
26        // the person to whom the vote is being delegated must be a valid voter
27        assert_eq!(delegate_opt.is_some(), true, "The delegated address is not a valid voter");
28
29        let delegate = delegate_opt.unwrap();
30
31        // the voter should not have already voted
32        if delegate.voted {
33            // If the delegate already voted,
34            // directly add to the number of votes
35            let voted_to = delegate.vote.unwrap() as usize;
36            self.proposals[voted_to].vote_count += sender_weight;
37        } else {
38            // If the delegate did not vote yet,
39            // add to her weight.
40            delegate.weight += sender_weight;
41    }
42}
```

```
41         }
42     }
43 }
```

You will see that in the delegation function above, we update the `sender.voted` and `sender.delegate` fields prior to checking if the person being delegated is a valid voter. The function will panic if the delegated person is not a valid voter and will roll back the changes made to the `sender.voted` and `sender.delegate` fields.

Your Turn!

This wraps up the tutorial. Practice what you learned with the following exercises:

- Update the `constructor` function so that if a vector of proposal names is provided, a new proposal object is created and added to `ballot.proposal`.
- Define the `give_voting_right` function as instructed.
- Add `vote` functionality and update the ballot according to the template requirements.
- Update `get_winning_proposal_name` functionality to return the name of the winning proposal.

Starting Point

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod ballot {
7      // use Hash
8      use ink_storage::collections::HashMap;
9      use ink_prelude::vec::Vec;
10     use ink_storage::traits::{PackedLayout, SpreadLayout};
11
12     // Structure to store Proposal information
13     #[derive(Clone, Debug, scale::Encode, scale::Decode, SpreadLayout)]
14     struct Proposal {
15         name: String,
16         vote_count: u32,
17     }
18
19     // Structure to store Voter information
20     #[derive(Clone, Debug, scale::Encode, scale::Decode, SpreadLayout)]
21     pub struct Voter {
22         weight: u32,
23         voted: bool,
24         delegate: Option<AccountId>,
25         vote: Option<i32>,
26     }
27
28     /// Defines the storage of your contract.
29     /// Add new fields to the below struct in order
30     /// to add new static storage fields to your contract.
31     #[ink(storage)]
32     pub struct Ballot {
33         chair_person: AccountId,
34         voters: HashMap<AccountId, Voter>,
35         proposals: Vec<Proposal>
36     }
37
38     impl Ballot {
39         #[ink(constructor)]
40         pub fn new(proposal_names: Option<Vec<String>>) -> Self {
41
42             // get chair person address
43             let chair_person = Self::env().caller();
44
45             // create empty proposal and voters
46             let mut proposals: Vec<Proposal> = Vec::new();
47             let mut voters = HashMap::new();
```

```

48
49         // initialize chair person's vote
50         voters.insert(chair_person, Voter{
51             weight: 1,
52             voted: false,
53             delegate: None,
54             vote: None,
55         });
56
57
58         // ACTION : Check if proposal names are provided.
59         //           * If yes then create and push proposal objects
60
61
62     Self {
63         chair_person,
64         voters,
65         proposals,
66     }
67 }
68
69     /// default constructor
70 #[ink(constructor)]
71 pub fn default() -> Self {
72     Self::new(Default::default())
73 }
74
75
76 #[ink(message)]
77 pub fn get_chairperson(&self) -> AccountId {
78     self.chair_person
79 }
80
81
82
83 pub fn get_voter(&self, voter_id: AccountId) -> Option<&Voter> {
84     self.voters.get(&voter_id)
85 }
86
87 pub fn get_voter_count(&self) -> usize {
88     self.voters.len() as usize
89 }
90
91     /// the function adds the provided voter id into possible
92     /// list of voters. By default the voter has no voting right,
93     /// the contract owner must approve the voter before he can cast
94 #[ink(message)]
95 pub fn add_voter(&mut self, voter_id: AccountId) -> bool {
96
97     let voter_opt = self.voters.get(&voter_id);
98     // the voter does not exists
99     if voter_opt.is_some() {

```

```
100             return false
101         }
102
103         self.voters.insert(voter_id, Voter{
104             weight:0,
105             voted:false,
106             delegate: None,
107             vote: None,
108         });
109         return true
110     }
111
112
113
114     /// given an index returns the name of the proposal at that index
115     pub fn get_proposal_name_at_index(&self, index:usize) -> &String {
116         let proposal = self.proposals.get(index).unwrap();
117         return &proposal.name
118     }
119
120     /// returns the number of proposals in ballot
121     pub fn get_proposal_count(&self) -> usize {
122         return self.proposals.len()
123     }
124
125     /// adds the given proposal name in ballot
126     /// to do: check uniqueness of proposal,
127     pub fn add_proposal(&mut self, proposal_name: String){
128         self.proposals.push(
129             Proposal{
130                 name:String::from(proposal_name),
131                 vote_count: 0,
132             });
133     }
134
135     /// Give `voter` the right to vote on this ballot.
136     /// Should only be called by `chairperson`.
137     #[ink(message)]
138     pub fn give_voting_right(&mut self, voter_id: AccountId) {
139         let caller = self.env().caller();
140         let voter_opt = self.voters.get_mut(&voter_id);
```

✓Potential Solution

```
1  #![cfg_attr(not(feature = "std"), no_std)]
2
3  use ink_lang as ink;
4
5  #[ink::contract]
6  mod ballot {
7      // use Hash
8      use ink_storage::collections::HashMap;
9      use ink_prelude::vec::Vec;
10     use ink_storage::traits::{PackedLayout, SpreadLayout};
11
12     // Structure to store Proposal information
13     #[derive(Clone, Debug, scale::Encode, scale::Decode, SpreadLayout)]
14     struct Proposal {
15         name: String,
16         vote_count: u32,
17     }
18
19     // Structure to store Voter information
20     #[derive(Clone, Debug, scale::Encode, scale::Decode, SpreadLayout)]
21     pub struct Voter {
22         weight: u32,
23         voted: bool,
24         delegate: Option<AccountId>,
25         vote: Option<i32>,
26     }
27
28     /// Defines the storage of your contract.
29     /// Add new fields to the below struct in order
30     /// to add new static storage fields to your contract.
31     #[ink(storage)]
32     pub struct Ballot {
33         chair_person: AccountId,
34         voters: HashMap<AccountId, Voter>,
35         proposals: Vec<Proposal>
36     }
37
38     impl Ballot {
39         #[ink(constructor)]
40         pub fn new(proposal_names: Option<Vec<String>>) -> Self {
41
42             // get chair person address
43             let chair_person = Self::env().caller();
44
45             // create empty proposal and voters
46             let mut proposals: Vec<Proposal> = Vec::new();
47             let mut voters = HashMap::new();
```

```

48
49         // initialize chair person's vote
50         voters.insert(chair_person, Voter{
51             weight:1,
52             voted:false,
53             delegate: None,
54             vote: None,
55         });
56
57
58         // ACTION : Check if proposal names are provided.
59         //           * If yes then create and push proposal objects
60         // if proposals are provided
61         if proposal_names.is_some() {
62             // store the provided propsal names
63             let names = proposal_names.unwrap();
64             for name in &names {
65                 proposals.push(
66                     Proposal{
67                         name: String::from(name),
68                         vote_count: 0,
69                     });
70                 }
71             }
72
73             Self {
74                 chair_person,
75                 voters,
76                 proposals,
77             }
78         }
79
80         /// default constrcutor
81 #[ink(constructor)]
82 pub fn default() -> Self {
83     Self::new(Default::default())
84 }
85
86
87 #[ink(message)]
88 pub fn get_chairperson(&self) -> AccountId {
89     self.chair_person
90 }
91
92
93
94 pub fn get_voter(&self, voter_id: AccountId) -> Option<&Voter> {
95     self.voters.get(&voter_id)
96 }
97
98 pub fn get_voter_count(&self) -> usize{
99     self.voters.len() as usize

```

```
100     }
101
102     /// the function adds the provided voter id into possible
103     /// list of voters. By default the voter has no voting right,
104     /// the contract owner must approve the voter before he can c
105     #[ink(message)]
106     pub fn add_voter(&mut self, voter_id: AccountId) -> bool{
107
108         let voter_opt = self.voters.get(&voter_id);
109         // the voter does not exists
110         if voter_opt.is_some() {
111             return false
112         }
113
114         self.voters.insert(voter_id, Voter{
115             weight:0,
116             voted:false,
117             delegate: None,
118             vote: None,
119         });
120         return true
121     }
122
123
124
125     /// given an index returns the name of the proposal at that i
126     pub fn get_proposal_name_at_index(&self, index:usize) -> &Str
127         let proposal = self.proposals.get(index).unwrap();
128         return &proposal.name
129     }
130
131     /// returns the number of proposals in ballot
132     pub fn get_proposal_count(&self) -> usize {
133         return self.proposals.len()
134     }
135
136     /// adds the given proposal name in ballot
137     /// to do: check unqiueness of proposal,
138     pub fn add_proposal(&mut self, proposal_name: String){
139         self.proposals.push(
140             Proposal{
141                 name:String::from(proposal_name),
```

Substrate

Exchange Integration

Please see the existing docs for Substrate integrations at the Polkadot Wiki. Questions? Join the Builders Guild chat at https://t.me/edg_developers

Docs on Integration <https://wiki.polkadot.network/docs/en/build-integration#docsNav>

Edgeware Runtime

Balances

This section will cover how Edgeware accounts and balances and how they are represented on-chain.

Address Format

Edgeware accounts each have an `AccountID`. The address format used in Substrate-based chains is SS58. SS58 is a modification of Base-58-check from Bitcoin with some minor modifications. All Edgeware addresses will start with a **lowercase letter** like j, m, l, i.

Addresses belonging to a particular Substrate-based chain are identified by an *address type* prefix. For Edgeware, this prefix is 7.

You can find other network prefixes [here](#), here are some notable ones:

- Polkadot addresses always start with the number 1.
- Kusama addresses always start with a capital letter like C, D, F, G, H, J...
- Generic Substrate addresses start with 5.

It's important to understand that the different formats for different networks are merely different representations of the same public key in a private-public keypair generated by an address generation tool. While it's currently recommended that you use different public-private keypairs across chains, it's possible to reuse them. Addresses are compatible across Substrate-based chains as long as you convert the format. More information on address portability within Substrate and Polkadot networks can be found [here](#).

Balances Pallet

The balances pallet defines EDG, the native token for Edgeware. More specifically, it defines storage items that track the tokens a user has, functions that users can call to transfer and manage those tokens, APIs which allow other modules to burn or mint those tokens, and hooks which allow other pallets to trigger functions when a user's balance changes.

The balances pallet is used throughout the Edgeware Runtime. Allowing the chain to keep track of governance and council votes, treasury balances, earning staking rewards, and depositing EDG into the contract and **EVM** pallet to run permissionless smart contracts.

Existential Deposit

When you generate an account (address), you only generate a *key* that lets you access it. The account does not exist yet on-chain. For that, it needs the existential deposit: 0.001 EDG. At current market prices, this means that almost anyone around the world can have an Edgeware account.

Having an account go below the existential deposit causes that account to be *reaped*. The account will be wiped from the blockchain's state to conserve space, along with any funds in that address. You do not lose access to the reaped address - as long as you have your private key or recovery phrase, you can still use the address - but it needs a top-up of another existential deposit to be able to interact with the chain.

Here's another way to think about existential deposits. Ever notice those `Thumbs.db` files on Windows or `.DS_Store` files on Mac? Those are junk, they serve no specific purpose other than making previews a bit faster. If a folder is completely empty save for such a file, you can remove the folder to clear junk off your hard drive. That does not mean you lose access to this folder forever - you can always recreate it. You have the *key*, after all - you're the computer's owner. It just means you want to keep your computer clean until you maybe end up needing this folder again, and then recreate it. Your address is like this folder - it gets removed from the chain when nothing is in it, but gets put back when it has at least the existential deposit.

Balance Reservations

When interactions with other pallets occur, certain reservations may be incurred by an individual's balance. A few terms are useful to define.

Total : The total amount of EDG held by any address.

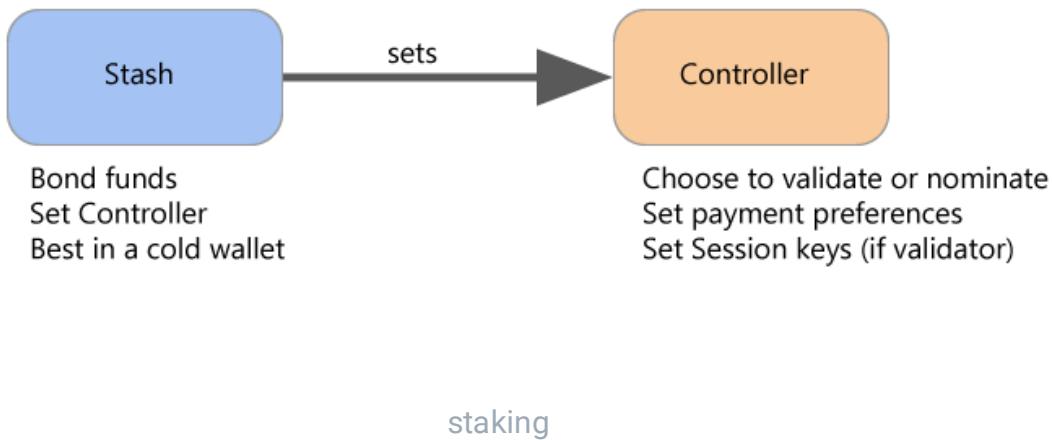
Free : The total amount of EDG that a user is able to transfer to another account. We get this by subtracting **Total** from any **reserved** tokens.

Reserved : The amount of EDG that has been used by other pallets. This would include voting for governance votes as well as participating in staking. While these tokens are still owed by an account holder, they may be slashed by another subsystem or pallet. For example, when a user nominates a validator they stake tokens for the opportunity to validate transactions. In the balances pallet, their tokens will be **reserved**, in the event of slashing, this amount may be deducted from the user's balance.

Staking Accounts: Stash and Controller

For Edgeware and many other PoS smart contract platforms, staking is the action that ensures the economic security of the blockchain. Users stake their tokens to a validator, who creates new blocks. For this effort, validators and nominators earn more tokens in the form of fees and inflationary rewards.

To participate in staking in Edgeware, a user is required to create and set multiple accounts for staking funds: `stash` and `Controller`. This allows a user to separately manage their tokens in cold storage for the `stash` account, and a `controller` account.



- **Stash:** This account holds funds bonded for staking, but delegates some functions to a Controller. As a result, you may actively participate with a Stash key kept in a cold wallet, meaning it stays offline all the time. You can also designate a Proxy account to vote in [governance](#) proposals.
- **Controller** This account acts on behalf of the Stash account, signaling decisions about nominating and validating. It sets preferences like payout account and commission. If you are a validator, it also sets your [session keys](#). It only needs enough funds to pay transaction fees.

We designed this hierarchy of separate key types so that validator operators and nominators can protect themselves much better than in systems with only one key. As a rule, you lose security anytime you use one key for multiple roles, or even if you use keys related by derivation. You should never use any account key for a "hot" session key in particular.

For Edgeware, and specifically for Session Keys and validation, we use the ed25519 cryptographic curve. Controller and Stash account key are recommended to be in this format also. For more on how keys are used in Substrate and the cryptography behind it [see here](#).

```
1 -> SessionKeys {
2     SessionKeys {
3         grandpa: ed25519_keyring.to_owned().public().into(),
4         aura: ed25519_keyring.to_owned().public().into(),
5         im_online: ed25519_keyring.to_owned().public().into(),
6         authority_discovery: sr25519_keyring.to_owned().public().into(),
```

For more on how keys are used in Substrate and the cryptography behind it [see here](#).

Credit: <https://wiki.polkadot.network/docs/en/learn-staking>

Consensus

Edgeware uses nominated proof-of-stake (NPoS), a relatively new type of scheme used to select the validators who are allowed to participate in the consensus protocol. We also explain the peculiar way in which validators get elected.

The NPoS scheme

This nominator-validator arrangement gives strong security guarantees. It allows for the system to select validators with massive amounts of aggregate stake – much higher than any single party's EDG holdings – and eliminate candidates with low stake. In fact, at any given moment we expect there to be a considerable fraction of all the EDG supply be staked in NPoS. This makes it very difficult for an adversarial entity to get validators elected (as they need to build a fair amount of **reputation** to get the required backing from nominators) and also very costly to attack the system if they do get elected (because any attack will result in large amounts of EDG being slashed).

- (i) "Slashing" is the loss or taking of some staked EDG and transferring it to the Edgeware Treasury.

The NPoS scheme is much more efficient than proof-of-work (PoW) and faster than standard proof-of-stake (PoS). Networks with deterministic finality must have a limited validator set (the size can be changed with governance). NPoS allows for virtually all EDG holders to continuously participate, thus maintaining high levels of security by putting more value at stake and allowing more people to earn a yield based on their holdings.

- (i) Through nomination, all EDG holders, not merely technically-capable validators, can participate in securing the network and benefitting from rewards.

Aura

Parameters

- ⓘ This page reflects the runtime [file of the chain viewable at Github](#), which is the final source of authority. You can also use the Polkadot.js apps to confirm the current value of these parameters on a testnet or mainnet at [Chain State tab > Constants > Select Parameter](#)

Accounts and Transactions

Parameter	Value	Description
Reaping Threshold	0.001 EDG	The minimum EDG required in the account balance to create or maintain an account.
Transaction Minimum	1e-18	The Min. amount you can send to an Edgeware Address.

Consensus

Parameter	Value
Consensus Mechanism	AURA
Finality Gadget	GRANDPA

Time

Exact Runtime code specifying constants related to block time and production.

Edgeware	Time	Slots*
Block	6 seconds	1
Epoch	60 minutes	600
Session	60 minutes (600 Blocks)	600
Era	6 hours (6 sessions)	3600

(i) This code is from <https://github.com/hicommonwealth/edgeware-node/blob/master/node/runtime/src/constants.rs>

It may be out of date on this page, refer to the link above for most up to date runtime data.

```

1  pub const MILLISECS_PER_BLOCK: Moment = 6000;
2  pub const SECS_PER_BLOCK: Moment = MILLISECS_PER_BLOCK / 1000;
3
4  pub const SLOT_DURATION: Moment = MILLISECS_PER_BLOCK;
5
6  // 1 in 4 blocks (on average, not counting collisions) will be primary
7  pub const PRIMARY_PROBABILITY: (u64, u64) = (1, 4);
8
9  pub const EPOCH_DURATION_IN_BLOCKS: BlockNumber = 1 * HOURS;
10 pub const EPOCH_DURATION_IN_SLOTS: u64 = {
11     const SLOT_FILL_RATE: f64 = MILLISECS_PER_BLOCK as f64 / SLOT_DURATION;
12
13     (EPOCH_DURATION_IN_BLOCKS as f64 * SLOT_FILL_RATE) as u64
14 };
15
16 // These time units are defined in number of blocks.
17 pub const MINUTES: BlockNumber = 60 / (SECS_PER_BLOCK as BlockNumber);
18 pub const HOURS: BlockNumber = MINUTES * 60;
19 pub const DAYS: BlockNumber = HOURS * 24;

```



hicommunity/edgeware-node

<https://github.com/hicommunity/edgeware-node/blob/master/node/runtime/src/constants.rs>

Staking

Parameter	Value	Description
Validator Slots	110	The total number of slots for active validation.
Validator Bonding Duration	28 hours	How long until you can unbond your funds after staking
Slash Deferral Duration	28 Eras (7 days)	Prevents overslashing and validators "escaping" and getting their nominators slashed with no repercussions to themselves
Slash Cancellation Vote	Requires 3/4 of Council to Approve	
Validator Term Duration		The time for which a validator is in the set after being elected. Note, this duration can be shortened in the case that a validator misbehaves.
Nomination Period		Countdown until a new validator set is elected according to Phragmen's method.

Democracy (Referenda)

Parameter	Value	Description

Launch Period	7 days	How long the public can select which proposal to hold a referendum on. i.e., Every week, the highest-weighted proposal will be selected to have a referendum
Voting Period	7 days	How long the public can vote on a referendum.
Enactment Delay	8 days	Time it takes for a successful referendum to be implemented on the network.
Passing Vote Criteria	Supermajority to pass	
Fast Track Voting Period	3 days	Minimum voting period allowed for an emergency referendum.
Veto	Not active	
Proposal Cancellation	2/3 of council to Approve	
Vote Cancellation	Cancellation	
Cool-off Period after Proposal Cancellation	7 days	The time a veto from the technical committee lasts before the proposal can be submitted again.
Min. EDG Deposit to Vote	100 EDG	

Vote Weighting by Lock Time

The schedule of weight boosts on a quadratic curve - meaning that exponentially increasing locktimes are required to achieve lesser proportional boosts in weight.

Vote Weight	Locktime
0.1x	None
1x	1x Enactment Period (8 days)

2x	2x Enactment Period (16 days)
3x	4x Enactment Period (32 days)
4x	8x Enactment Period (64 days)
5x	16x Enactment Period (128 days)
6x	32x Enactment Period (256 days)

Council Elections

Parameter	Value	Description
Term Duration	28 days	The length of a council member's term until the next election round. A new councilperson is elected every 28 days based on the Phragmen algorithm. If the term duration is changed the current term is affected when <code>BlockNumber % TermDuration == 0</code> , upon which a new council (or councilperson?) will be selected.
Candidacy Bond	1000 EDG	The amount a user must bond to submit their candidacy.
Voting Bond	10 EDG	The amount of EDG that a voter must lock to vote for Council.
Council Member Slots	13 members	The size of the council.
Runners-up Slots	7 members	The number of slots that will be displayed as a runner-up.
Council Voting Period	??	The council's voting period for motions.

Treasury

Parameter	Value	Description
Budgeting Period	7 days	When the treasury can spend again after spending previously.
Proposal Bond	5% and minimum 1000 EDG	The amount required to bond in order to propose a treasury spend. If approved, it is returned, if the proposal fails, it is burnt.
Burn unspent treasury funds	Off	This deactivates a burn of all unspent treasury funds at the end of a budgeting period.

Signaling

Parameter	Value	Description
Signaling Period	7 days	The length of time a signal proposal is active for engagement.
Signaling Proposal Bond	100 EDG	The amount of EDG required to bond to submit a signaling proposal.

Identity

Parameter	Value	Description
Required Bond Per Identity	10 EDG	Bond required to store IDs on-chain.
Required Bond Per Each Additional Identity Field	2.5 EDG	Bond required to store additional IDs on-chain Beyond Legal Name.
Sub-Account Deposit	2 EDG	Amount required to deposit in order to create a sub account.
Maximum Sub-Accounts	100	The maximum number of sub account an account may have.

Economics

Parameter	Value (Units)	Description
Minimum Inflation	.025	The min. inflation rate the system will permit.
Maximum Inflation	 0.1	The max. inflation rate the system will permit.
Ideal Staking Rate	0.8	The ideal proportion of EDG tokens staked compared to total EDG supply.
Falloff (Decay) Rate	 0.05	The decay rate on inflation when the actual staking rate becomes greater than the ideal staking rate.
Max Piece Count	40	

Test Precision	0.005
Ideal Interest Rate	12.5% The ideal returns that an individual validator earns.

Contract configuration

Parameter	Value	Description
Contract Transfer Fee	0.10 EDG	****
Contract Creation Fee	0.10 EDG	****
Contract Transaction Base Fee	0.10 EDG	****
Contract Transaction Byte Fee	0.001 EDG	****
Contract Fee	0.10 EDG	****
Tombstone Deposit	1 EDG	****
Rent Byte Fee	1 EDG	****
Rent Deposit Offset	1000 EDG	****
Surcharge Reward	150 EDG	

Genesis

Parameter	Value
Council Members	1 Member at time of Genesis (Commonwealth Labs 0x02456...)

Identity Attestation Provider	1 Verifier at time of Genesis (Commonwealth Labs 0x92c32...)
-------------------------------	---

Economics

Token Economics Information

	Value	Info
Total Initial Supply		5 Billion
Total Planned Inflation	The total amount of EDG minted will remain the same year after year, causing the percentage inflation to be disinflationary , with yearly inflation falling to approximately 16.6% in the second year and so on.	
Maximum Token Supply		Variable by governance action.
Inflation Rate	Dependent upon the staking rate in the network, but maxes out at 20% as the staking ratio approaches 80% currently. This latter will likely shift to 50% through governance (the 'ideal staking rate')	
Maximum Stake		No maximum
Minimum Stake		No minimum
Unbonding / Undelegating period	6 Hours currently (unverified) , but should become several days after an upcoming upgrade.	
Token Type	Native Platform Token (like ETH)	
Ticker	EDG	
Decimal Places	18	
Inflation per Block	~95 EDG	

Ideal	
Inflation Rate	158 EDG per block

Consensus

Edgeware uses Nominated Proof of Stake (NPoS) as its consensus method. There is a **known and limited** number of validators in the active set and this active set number is decided by governance. Inclusion in the active set is determined by your total self-bonded and delegated stake. The minimum stake to get in the active set varies daily and depends upon the number of validators attempting to be included and the amount of stake on each. In a NPoS system, each elected validator has equal say in consensus and is rewarded equally, not according to the proportion of stake, as a result it is incentivized for a validator to distribute their stake and run multiple validator nodes— not only among their own nodes but potentially among other validators to normalize stake across the set, something NPoS does to help prevent centralized stake.

-  No names have been assigned for fractions of an EDG, we refer to them by the default dollars/cents (1 EDG is a dollar, one EDG cent is 0.01 EDG.) This is a good opportunity for a proposal.

Staking Estimator Spreadsheet

Inflation

Inflation is currently ~95 EDG / block.

The genesis specification parameters (in decimal) related to this ideal condition, and what the algorithm uses are:

Parameter	Value
Minimum Inflation	0.025

Maximum Inflation	0.125
Ideal Staking Rate	0.800
Decay Rate (aka Falloff)	0.050

Token Uses

On Edgeware, EDG has multiple functions.

- Staking
- Voting (Governance, elections, proposal making.)
- Staking Delegation
- Voting Delegation
- Pay transaction fees for state changes in smart contracts.

Token Supply Chart

See and edit the full sheet.



Edgeware Supply (with graph)

[https://docs.google.com/spreadsheets/d/
1bKuD0GnQr-HZlrPdos6UuVBfOu27MxAuKqF5t2lIOHM
/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1bKuD0GnQr-HZlrPdos6UuVBfOu27MxAuKqF5t2lIOHM/edit?usp=sharing)

Return Rate Calculator for Validators

The following graph lets you input network parameters and will provide an annualized yearly return rate for a single validator.

You can [click here](#) to see details, alter the parameters, and experiment with different values.

Transaction Fees

Transaction Fees

Several resources in a blockchain network are limited, for example, storage and computation. Transaction fees prevent individual users from consuming too many resources. Edgeware uses a weight-based fee model as opposed to a gas-metering model. As such, fees are charged prior to transaction execution; once the fee is paid, nodes will execute the transaction.

[Web3 Foundation Research](#) designed the Polkadot fee system with the following objectives:

- Each Relay Chain block should be processed efficiently to avoid delays in block production.
 - The growth rate of the Relay Chain should be bounded.
 - Each block should have space for special, high-priority transactions like misconduct reports.
 - The system should be able to handle spikes in demand.
 - Fees should change slowly so that senders can accurately predict the fee for a given transaction.
-

Fee Calculation

Fees on Edgeware are calculated based on three parameters:

- A per-byte fee (also known as the "length fee")
- A weight fee
- A tip (optional)

The length fee is the product of a constant per-byte fee and the size of the transaction in bytes.

Weights are a fixed number designed to manage the time it takes to validate a block. Each transaction has a base weight that accounts for the overhead of inclusion (e.g. signature verification) as well as a dispatch weight that accounts for the time to execute the

transaction. The total weight is multiplied by a per-weight fee to calculate the transaction's weight fee.

Tips are an optional transaction fee that users can add to give a transaction higher priority.

Together, these three fees constitute the inclusion fee. This fee is deducted from the sender's account prior to transaction execution. A portion of the fee will go to the block producer and the remainder will go to the Treasury. At Edgeware's genesis, this is set to 20% and 80%, respectively.

Block Limits and Transaction Priority

Blocks in Edgeware have both a maximum length (in bytes) and a maximum weight. Block producers will fill blocks with transactions up to these limits. A portion of each block - currently 25% - is reserved for critical transactions that are related to the chain's operation. Block producers will only fill up to 75% of a block with normal transactions. Some examples of operational transactions:

- Misbehavior reports
- Council operations
- Member operations in an election (e.g. renouncing candidacy)

Block producers prioritize transactions based on each transaction's total fee. Since a portion of the fee will go to the block producer, producers will include the transactions with the highest fees to maximize their reward.

Fee Adjustment

Transaction volume on blockchains is highly irregular, and therefore transaction fees need a mechanism to adjust. However, users should be able to predict transaction fees.

Edgeware uses a slow-adjusting fee mechanism with tips to balance these two considerations. In addition to block limits, Edgeware also has a block fullness target. Fees increase or decrease for the next block based on the fullness of the current block relative to

the target. The per-weight fee can change up to 30% in a 24 hour period. This rate captures long-term trends in demand, but not short-term spikes. To consider short term spikes, Edgeware uses tips on top of the length and weight fees. Users can optionally add a tip to the fee to give the transaction a higher priority.

Shard Transactions

The transactions that take place within Polkadot's shards - parachains and parathreads - do not incur Relay Chain transaction fees. Users of shard applications do not even need to hold DOT tokens, as each shard has its own economic model and may or may not have a token. There are, however, situations where shards themselves make transactions on the Relay Chain.

Parachains have a dedicated slot on the Relay Chain for execution, so their collators do not need to own DOT in order to include blocks. The parachain will make some transactions itself, for example, opening or closing an XCMP channel, participating in an auction to renew its slot, or upgrading its runtime. Parachains have their own accounts on the Relay Chain and will need to use those funds to issue transactions on the parachain's behalf.

Parathreads will also make all the same transactions that a parachain might. In addition, the collators need to participate in an auction every block to progress their chain. The collators will need to have DOT to participate in these auctions.

Other Resource Limitation Strategies

Transaction weight must be computable prior to execution, and therefore can only represent fixed logic. Some transactions warrant limiting resources with other strategies. For example:

- **Bonds:** Some transactions, like voting, may require a bond that will be returned or slashed after an on-chain event. In the voting example, returned at the end of the election or slashed if the voter tried anything malicious.
- **Deposits:** Some transactions, like setting an identity or claiming an index, use storage space indefinitely. These require a deposit that will be returned if the user decides to free storage (e.g. clear their ide).

- Burns: A transaction may burn funds internally based on its logic. For example, a transaction may burn funds from the sender if it creates new storage entries, thus increasing the state size.
 - Limits: Some limits are part of the protocol. For example, nominators can only nominate 16 validators. This limits the complexity of Phragmén.
-

Advanced

This page only covered transactions that come from normal users. If you look at blocks in a block explorer, though, you may see some "extrinsics" that look different from these transactions. In Edgeware, an extrinsic is a piece of information that comes from outside the chain. Extrinsic fall into three categories:

- Signed transactions
- Unsigned transactions
- Inherents

This page only covered signed transactions, which is the way that most users will interact with Edgeware. Signed transactions come from an account that has funds, and therefore Edgeware can charge a transaction fee as a way to prevent spam.

Unsigned transactions are for special cases where a user needs to submit an extrinsic from a key pair that does not control funds. For example, when users claim their DOT tokens after genesis, their DOT address doesn't have any funds yet, so that uses an unsigned transaction. Validators also submit unsigned transactions in the form of "heartbeat" messages to indicate that they are online. These heartbeats must be signed by one of the validator's session keys. Session keys never control funds. Unsigned transactions are only used in special cases because, since Edgeware cannot charge a fee for them, each one needs its own, custom validation logic.

Finally, inherents are pieces of information that are not signed or included in the transaction queue. As such, only the block author can add inherents to a block. Inherents are assumed to be "true" simply because a sufficiently large number of validators have agreed on them being reasonable. For example, Edgeware blocks include a timestamp inherent. There is no way to prove that a timestamp is true the way one proves the desire to send funds with a signature. Rather, validators accept or reject the block based on how reasonable they find the

timestamp. In Edgeware, it must be within some acceptable range of their own system clocks.

Resources

- Origin
- Substrate Weights
- Substrate Fees

Governance

A substantial majority (estimated 80%) of utility tokens and smart contract blockchains have indicated their intention to move to on-chain and decentralized governance. However, at the time of writing, there are few successful implementations of on-chain governance. The few systems that have launched to date are primarily on-chain treasuries, delegated proof-of-stake (DPoS) systems, and direct token-weighted voting models.

Directly token-weighted on-chain processes are generally easily captured by large economic holders in chains, such as block producers, miners, and exchanges. Additionally, very few governance systems have any level of active use today.

Communication, conversation, and voting are limited by the same unintuitive interfaces that hamper the growth of cryptocurrencies as a whole. Edgeware aims to be a pioneer network in developing effective on-chain governance.

By iterating upon product interfaces, voting systems, and other governance primitives, Edgeware can accelerate the implementation and deployment of core technologies like sharding, proof-of-stake, efficient SNARK implementations, and run-time changes, implementing technical advances at a faster pace than other blockchains. Edgeware network upgrades will be easier to coordinate and faster to deploy by using a clearly defined on-chain governance process.

Process

Edgeware stakeholders will be able to collaborate on the roadmap of the network through on-chain signaling, by holding informal votes for the inclusion of specific items and proposals. This planning stage is a natural precursor to the on-chain governance process, and formal votes are held to fund and upon which agreed upon features are implemented. Establishing robust procedures for conducting on-chain upgrades will be an essential step towards ensuring the security of the network. All network upgrades should be audited and tested by multiple independent parties, and additionally approved by a significant quorum of EDG holders.

Governance Parameters at Launch

- **Launch period:** 7 days
- **Voting period:** 7 days
- **Fast-track voting period:** 3 days
- **Veto:** None
- **Enactment delay:** 8 days
- **Cancellation:** 2/3 of council
- **Cooloff period after cancellation:** 7 days

Council

Council

Intro

Decentralized systems present a problematic scenario for online and active participation, often leading to low turnout. Therefore, council members can bias the quorum, the number of votes needed and the relative difference between affirmative and non-affirmative votes.

The Edgeware council module will eventually expand to allow 24 individual accounts to hold some exclusive rights over the network with council members having a fixed term of 12 months. At network launch, the number of council members will be thirteen. The council votes on a peer basis, that is, no coin-weighted votes.

Quorum biasing allows the council to change the effective supermajority required to make it easier or more difficult for a proposal to pass in the case that there is no clear majority of voting power backing it or against it. If all council members vote for a proposal, then the required amount of non-council member votes is lessened. The reverse also applies. When the council votes and more than one member dissents, then the quorum is negatively biased.

Council Elections

How Council Members are Selected by the System

The Phragmen method is also used in the council election mechanism. When you vote for council members, you can select up to 16 different candidates, and then place a reserved bond which is the weight of your vote. Phragmen will run once on every election to determine the top candidates to assume council positions and then again amongst the top candidates to equalize the weight of the votes behind them as much as possible.

→ [The Sequential Phragmen Method](#)

/edgeware-runtime/staking/the-sequential-phragmen-method

Historical Council Record

Council Elections

How Council Members are Selected by the System

The Phragmen method is also used in the council election mechanism. When you vote for council members, you can select up to 16 different candidates, and then place a reserved bond which is the weight of your vote. Phragmen will run once on every election to determine the top candidates to assume council positions and then again amongst the top candidates to equalize the weight of the votes behind them as much as possible.

→ [The Sequential Phragmen Method](#)

/edgeware-runtime/staking/the-sequential-phragmen-method

Council Operations and Resources

Treasury

When adopting a treasury proposal via Polkadot UI, the 'Send to Council' Button may present the opportunity to set a 'threshold', or it may hardcode a threshold, currently believed to be 8.. This is a threshold of agreement, so if the threshold is 8, then 8 councilpeople must approve it, and then, only if that is fulfilled, the logic checks the on-chain parameters for other thresholds, supermajority, etc. If that also passes, then the treasury proposal now council motion is accepted and processed.

Council Motions

- Motions to accept are separate from motions to reject, each is its own standalone proposal.
- Duplicate proposals are not accepted by the system while one is active.
- You can manipulate the threshold of Motions to reject to prevent easy rejections because a lower threshold version cannot be submitted, reason being that it is a duplicate. This may be considered a bug, exploit, or other unintended use.
- Threshold parameters can be set to force simple-majority or supermajority on motions that may normally need less consensus.
- When submitting a motion, you must ensure that the threshold for that action according to the runtime parameters, \leq the threshold you submit. For example:
 - **Fail Case** Councilor Thom submits a Treasury.rejectProposal motion with a threshold of 1. Thom's vote is automatically counted as affirmative because he submitted the proposal. The threshold of 1 is met, and the motion attempts to execute. However, the network runtime has a threshold of 2 for this action. The execution fails because the motion's job is done, but impermissible for the network, and nothing will go to the council.

Voting for Council

Intro

The council is an elected body of on-chain accounts that are intended to represent the passive stakeholders of Edgeware. The council has two major tasks in governance: proposing referenda and vetoing dangerous or malicious referenda. This guide will walk you through voting for councillors in the elections.

Voting for Councillors

Voting for councillors requires you to lock your EDG for the duration of your vote. Like the validator elections, you can approve up to 16 different councillors and your vote will be equalized among the chosen group. Unlike validator elections, there is no unbonding period for your reserved tokens. Once you remove your vote, your tokens will be liquid again.

Warning: It is your responsibility not to put your entire balance into the reserved value when you make a vote for councillors. It's best to keep *at least* a few KSM to pay for transaction fees.

Go to the [Polkadot Apps Dashboard](#), [connect to the Edgeware endpoint](#), and click on the "Council" tab. On the right side of the window there are two blue buttons, click on the one that says "Vote."



Since the council uses approval voting, when you vote you signal which of the validators you approve of and your voted tokens will be equalized among the selected candidates. Select up to 16 council candidates by moving the slider to "Aye" for each one that you want to be elected. When you've made the proper configuration submit your transaction.

Voting for council members¶

1. Using the [Polkadot UI](#), make sure you have an account and selected the Kusama network under the [settings tab](#).
2. Navigate to the [Council tab](#) to see current council candidates.

3. In the **Kusama forum**, you will find a thread dedicated to council members proposing their candidacy and find out more information.
4. Head over to the **Extrinsics tab**, select the account you wish to vote with, and select `council` under "submit the following extrinsic." Choose `setApprovals(votes, index)` in the second column, enter the council index of the candidate you wish to vote for, and select `yes` to support the candidate, and `nay` to vote against it.
5. Click `Submit transaction` and sign the transaction.



You should see your vote appear in the interface immediately after your transaction is included.

Removing your Vote

In order to get your reserved tokens back, you will need to remove your vote. Only remove your vote when you're done participating in elections and you no longer want your reserved tokens to count for the councillors that you approve.

Go to the "Extrinsics" tab on **Substrate Apps Dashboard**.

Choose the account you want to remove the vote of and select the "electionsPhragmen -> removeVoter()" options and submit the transaction.



When the transaction is included in a block you should have your reserved tokens made liquid again and your vote will no longer be counting for any councillors in the elections starting in the next term.

Run For Council

Intro

The council is an elected body of on-chain accounts that are intended to represent the passive stakeholders of the Edgeware network. The council has two major tasks in governance: proposing referenda and vetoing dangerous or malicious referenda. This guide will walk you through entering your candidacy to the council.

Submit Candidacy

Submitting your candidacy for the council requires a bond of 1000 EDG.

 **Parameter Note:** The Candidacy Bond, or the current minimum bond to submit candidacy is 1000 EDG.

The bond will be forfeited if your candidacy does not win or become a runner-up, but if you become a member of the council you will eventually get your bond back. Runner-ups are selected after every round and are reserved members in case one of the winners gets forcefully removed.

It is a good idea to announce your council intention before submitting your candidacy so that your supporters will know when they can start to vote for you. You can also vote for yourself in case no one else does.

Go to [Polkadot Apps Dashboard](#), [connect to the Edgeware endpoint](#), and navigate to the "Council" tab. Click the button on the right that says "Submit Candidacy."

Steps to run for Council¶

1. Using the [Polkadot UI](#), make sure you have an account and selected the Kusama network under the [settings tab](#).

2. Navigate to the **Council tab** to see current council candidates.
3. In the **Kusama forum**, you will find a thread dedicated to council members proposing their candidacy and find out more information. Add your account number, reasons for being a suitable council member and any further information you want to share.
4. Head over to the **Extrinsics tab**, select the account you wish to vote with, and select `council` under "submit the following extrinsic." Choose `submitCandidacy(slot)` in the second column and select the slot you prefer to be in.
5. Click `Submit transaction` and sign the transaction.



a

After making the transaction, you will see your account appear on the right column under "Candidates."



b

It is a good idea now to lead by example and give yourself a vote.

Voting on Candidates

Next to the button to submit candidacy is another button titled "Vote." You will click this button to make a vote for yourself (optional).



c

The council uses the **Phragmen** approval voting which is also used in the validator elections. This means that you can choose up to 16 distinct candidates to vote for and your stake will equalize between them. For this guide, choose to approve your own candidacy by clicking on the switch next to your account and changing it to say "Aye."



d

Winning an Election

If you are one of the lucky ones to win a council election you will see your account move to the left column under the heading "Members."



e

Congratulations! Now you are able to participate on the council by making motions or vetoing proposals. It's a good idea to now add an Identity so that others know who the account belongs to and [join the Commonwealth Edgeware forum](#).

Signaling

Intro

The signaling module allows users to vote to create non-binding polls. Non-binding polls are an important part of the governance process for existing 10 blockchains and have been administered previously in an ad-hoc manner.

Users should natively be able to create and distribute polls for signaling interest in different proposals, strategies, and directions. Individuals can choose between many different methods for voting on a proposal (e.g., binary or ranked choice) and moreover allows for delegated voting—improving the total stake allocated towards a vote. Once a direction has been chosen, the council or the democracy module can create a binding on-chain referendum.

Module Details

This module enables one to create signaling proposals and vote on them. This is useful for engaging parts of the community and understanding how the community reacts to a given idea before putting it forth in a state-changing proposal through the main governance mechanism.

The lifecycle for using this module is:

1. Create signaling proposals
2. Vote on signaling proposals
3. Engage in off-chain discussion

Democracy

..From the Edgeware Whitepaper

The democracy module creates and moves different binding referendum along the voting process.

Steps

- Proposing Referenda (Involved info: [Referenda](#))
- Voting for a proposal (Involved info: [Voluntary Locking](#))
- Tallying (Involved info: [Adaptive Quorum Biasing](#))

Referendums allows for the execution of an arbitrary extrinsic call on the chain as the root user.

This allows for execution of rather significant actions (e.g. setting balance, modifying timestamps, etc) without performing a runtime upgrade.

The module currently only supports binary supermajority-to-pass votes unless submitted by a council member (which also supports supermajority-to-fail and simple majority).

Individuals vote on binary choice ballots that are tallied by coin-weight with lock-time adjustments. At the outset, all votes will be restricted to a two-week fixed period. In the future, the democracy module will be extended to many [VotingTypes](#).

→ [Democracy Features](#)

/edgeare-runtime/governance/democracy/democracy-features

Guides to Referenda

The public referenda chamber is one of the two bodies of on-chain governance in Edgeware. The other body is the [council](#).

1. Public referenda can be proposed and voted on by **any token holder in the system as long as they provide a bond.**
2. After a proposal is made, others can agree with it by *seconding* it and putting up tokens equal to the original bond.
3. Every launch period, the most seconded proposal will be moved to the public referenda table for active voting.

Voters who are willing to lock up their tokens for a greater duration of time can do so and get their vote amplified. For more details on the governance system please see [here](#).

-  This guide will instruct token holders how to propose, delegate votes, and vote on public referenda using the Democracy module as it's implemented in Edgeware using [the Polkadot UI](#).

Important Parameters

The important parameters to be aware of when voting using the Democracy module are as follow:

-  These parameters can change based on governance. Refer to [Network Parameters](#) or the Node Runtime for the the most authoritative reference.

Launch Period - How often new public referenda are launched.

Parameter	Value	Description
-----------	-------	-------------

Launch Period	7 days	How often new public referenda are selected by seconding actions and launched to the public for active vote.
Voting Period	7 days	How often votes for referenda are tallied.
Emergency Voting Period	3 days	The minimum voting period for a fast-tracked emergency referendum. (aka Fast Track)
Minimum Deposit	100 EDG	The minimum amount to be used as a deposit for a public referendum proposal.
Enactment Period	8 days	The minimum time period for locking funds <i>and</i> the period between a proposal being approved and enacted
Cool off Period	7 days	The time period where a proposal may not be re-submitted after being vetoed.

Proposing an Action

Proposing an action to be taken requires you to bond some tokens. In order to ensure you have enough tokens to make the minimum deposit you can check the parameter in the chain state.

On Polkadot Apps you can use the "Democracy" tab to make a new proposal. In order to submit a proposal, you will need to submit what's called the preimage hash. The preimage hash is simply the hash of the proposal to be enacted. The easiest way to get the preimage hash is by clicking on the "Submit preimage" button and configuring the action that you are proposing.

For example, if you wanted to propose that the account "Dave" would have a balance of 10 tokens your proposal may look something like the below image. The preimage hash would be `0xa50af1fadfc818feea213762d14cd198404d5496bca691294ec724be9d2a4c0`. You can

copy this preimage hash and save it for the next step. There is no need to click Submit Preimage at this point, though you could. We'll go over that in the next section.

Submit preimage

send from account ?
FERDIE transferrable 1.000G DEV
5CiPPseXPECbkjWCa6MnjNokrgYjMqmKndv2rSnekmsK... ▾

propose ?
balances setBalance(who, new_free, new_reserved)
Set the balances of a given account. ▾

who: LookupSource
DAVE 5DAAnrj7VHTznn2AwBemMuyBwZws6FNFjdyVXUeYum... ▾

new_free: Compact<Balance>
10 DEV ▾

new_reserved: Compact<Balance>
0 DEV ▾

preimage hash ?
0xa50af1fadfc818feea213762d14cd198404d5496bca691294ec724be9d2a4c0

imminent preimage (proposal already passed)

× Cancel or + Submit preimage

submit preimage

Now you will click on the "Submit proposal" button and enter the preimage hash in the input titled "preimage hash" and *at least* the minimum deposit into the "locked balance" field. Click on the blue "Submit proposal" button and confirm the transaction. You should now see your proposal appear in the "proposals" column on the page.

Submit proposal

send from account ?
FERDIE transferrable 999.899M DEV
5CiPPseXPECbkjWCa6MnjNokrgYjMqmKndv2rSnekmsK... ▾

preimage hash ?
0xa50af1fadfc818feea213762d14cd198404d5496bca691294ec724be9d2a4c0

locked balance ?
10000d DEV ▾

× Cancel or + Submit proposal

submit proposal

Now your proposal is visible by anyone who accesses the chain and others can second it or submit a preimage. However, it's hard to tell what exactly this proposal does since it shows the hash of the action. Other holders will not be able to make a judgement for whether they second it or not until someone submits the actual preimage for this proposal. In the next step you will submit the preimage.

proposals

0	 FERDIE	Locked	proposal hash	<input type="button" value="Second"/>	<input type="button" value="or"/>	<input type="button" value="Preimage"/>
100,000,000 DEV			Oxa50af1fadfc818feea213762d14cd198404d5496bca691294ec724be9d2a4c0			

proposals

Submitting a Preimage

The act of making a proposal is split from submitting the preimage for the proposal since the storage cost of submitting a large preimage could be pretty expensive. Allowing for the preimage submission to come as a separate transaction means that another account could submit the preimage for you if you don't have the funds to do so. It also means that you don't have to pay so many funds right away as you can prove the preimage hash out-of-band.

However, at some point before the proposal passes you will need to submit the preimage or else the proposal cannot be enacted. The guide will now show you how to do this.

Click on the blue "Submit preimage" button and configure it to be the same as what you did before to acquire the preimage hash. This time, instead of copying the hash to another tab, you will follow through and click "Submit preimage" and confirm the transaction.

Submit preimage

send from account [?](#)
FERDIE

transferrable 1.000G DEV
5CiPPseXPECbkjWCa6MnjNokrgYjMqmKndv2rSnekSK... ▾

propose [?](#)
balances

setBalance(who, new_free, new_reserved)
Set the balances of a given account. ▾

who: LookupSource
DAVE

5DAAnrj7VHTznn2AwBemMuyBwZWs6FNFjdyVXUeYum... ▾

new_free: Compact<Balance>
10

DEV ▾

new_reserved: Compact<Balance>
0

DEV ▾

preimage hash [?](#)
0xa50af1fadfc818feea213762d14cd198404d5496bca691294ec724be9d2a4c0

imminent preimage (proposal already passed)

[✖ Cancel](#) or [+ Submit preimage](#)

submit preimage

Once the transaction is included you should see the UI update with the information for your already submitted proposal.

proposals

0		FERDIE	locked	100,000,000 DEV	balances.setBalance	Second
					► Set the balances of a given account. This will alter 'FreeBalance' and 'ReservedBalance' in storage. It will also decrease the total issuance of the system ('TotalIssuance'). If the new free or reserved balance is below the existential deposit, it will reset the account nonce ('frame_system::AccountNonce'). The dispatch origin for this call is 'root'. # <weight> - Independent of the arguments. - Contains a limited number of reads and writes. # </weight>	

proposals updated

Seconding a Proposal

Seconding a proposal means that you are agreeing with the proposal and backing it with an equal amount of deposit as was originally locked. By seconding a proposal you will move it higher up the rank of proposals. The most seconded proposal - in value, not number of supporters - will be tabled as a referendum to be voted on every launch period.

To second a proposal, navigate to the proposal you want to second and click on the "Second" button.

proposals

A screenshot of a proposal details page. At the top right is a blue button labeled "Second". Below the button, there is a detailed description of the proposal:

0 FERDIE locked 100,000,000 DEV balances.setBalance
► Set the balances of a given account. This will alter `FreeBalance` and `ReservedBalance` in storage. It will also decrease the total issuance of the system (`TotalIssuance`). If the new free or reserved balance is below the existential deposit, it will reset the account nonce (`frame_system::AccountNonce`). The dispatch origin for this call is `root`. # <weight> - Independent of the arguments. - Contains a limited number of reads and writes. # </weight>

second button

You will be prompted with the full details of the proposal (if the preimage has been submitted!) and can then broadcast the transaction by clicking the blue "Second" button.

Second proposal

#0: balances.setBalance

▼ Set the balances of a given account. This will alter `FreeBalance` and `ReservedBalance` in storage. It will also decrease the total issuance of the system (`TotalIssuance`). If the new free or reserved balance is below the existential deposit, it will reset the account nonce (`frame_system::AccountNonce`). The dispatch origin for this call is `root`. # <weight> - Independent of the arguments. - Contains a limited number of reads and writes. # </weight>

The second proposal details page shows the transaction fields:

who: LookupSource DAVE 5DAAnrj7VHTznn2AwBemMuyBwZws6FNF...

new_free: Compact<Balance> 10.000 DEV

new_reserved: Compact<Balance> 0 DEV

The second proposal details page shows a dropdown for selecting an account:

second with account BOB 5FHnew46xGXgs5mUiveU4sbTyGBzmstUs... ▾

✖ Cancel or ➡ Second

second confirm

Once successful you will see your second appear in the dropdown in the proposal details.

0	 FERDIE	locked 100,000,000 DEV	balances.setBalance ► Set the balances of a given account. This will alter 'FreeBalance' and 'ReservedBalance' in storage. It will also decrease the total issuance of the system ('TotalIssuance'). If the new free or reserved balance is below the existential deposit, it will reset the account nonce ('frame_system::AccountNonce'). The dispatch origin for this call is 'root'. # <weight> - Independent of the arguments. - Contains a limited number of reads and writes. # </weight>	▼ Seconds (1)  BOB	⌚ Second
---	--	---------------------------	--	---	----------

second result

Voting on a Proposal

Prerequisites to Vote:

- Have an EDG Account
- Have EDG Tokens
- Be prepared to have **your EDG locked** during the duration of the referenda. (Voting Period is 7 days)
- Additionally, If you vote `aye` ('approve', 'yes', 'in support') for a proposal and it passes, your tokens will be **locked** until the proposal has been enacted (another 7 days.)



Voting locks all of the tokens in the account you use to vote, and weights your votes accordingly. In order to avoid having your entire EDG supply locked, create a separate voting account and transfer the amount you wish to vote with, over to that account first.

At the end of each launch period, the most seconded proposal will move to referendum. During this time you can cast a vote for or against the proposal. You may also lock up your tokens for a greater length of time to weigh your vote more strongly. During the time your tokens are locked, you are unable to transfer them, however they can still be used for further votes. Locks are layered on top of each other, so an eight week lock does not become a 15 week lock if you vote again a week later, rather another eight week lock is placed to extend the lock just one extra week.

To vote on a referendum, navigate to the "[Democracy](#)" tab of [Polkadot Apps](#). Any active referendum will show in the "referenda" column. Click the blue button "Vote" to cast a vote for the referendum.

If you would like to cast your vote for the proposal select the "Aye, I approve" option. If you would like to cast your vote against the proposal in referendum you will select "Nay, I do not approve" option.

The second option is to select your conviction for this vote. The longer you are willing to lock your tokens, the stronger your vote will be weighted. Unwillingness to lock your tokens means that your vote only counts for 10% of the tokens that you hold, while the maximum lock up of 256 days means you can make your vote count for 600% of the tokens that you hold.

When you are comfortable with the decision you have made, click the blue "Vote" button to submit your transaction and wait for it to be included in a block.

Vote on proposal

#28: identity.addRegistrar

► Add a registrar to the system. The dispatch origin for this call must be `RegistrarOrigin` or `Root`. - `acco...

vote with account ?
LMS

record my vote as ?
Aye, I approve

ELmaX1aPkyEF7TSmYbbyCjmSgrBpGHv9E... ▾

0.1x voting balance, no lockup period

0.1x voting balance, no lockup period

1x voting balance, locked for 1x enactment (8.00 days)

2x voting balance, locked for 2x enactment (16.00 days)

3x voting balance, locked for 4x enactment (32.00 days)

4x voting balance, locked for 8x enactment (64.00 days)

5x voting balance, locked for 16x enactment (128.00 days)

voting

Delegate a Vote

If you are too busy to keep up and vote on upcoming referenda, there is an option to delegate your vote to another account whose opinion you trust. When you delegate to another account, that account gets the added voting power of your tokens along with the conviction that you set. The conviction for delegation works just like the conviction for regular voting, except your tokens may be locked longer than they would normally since locking resets when you undelegate your vote.

The account that is being delegated to does not make any special action once the delegation is in place. They can continue to vote on referenda how they see fit. The difference is now when the Democracy system tallies votes, the delegated tokens now are added to whatever vote the delegatee has made.

You can delegate your vote to another account and even attach a "Conviction" to the delegation. Navigate to the "Extrinsics" tab on Polkadot Apps and select the options "democracy" and "delegate". This means you are accessing the democracy pallet and choosing the delegate transaction type to send. Your delegation will count toward whatever the account you delegated for votes on until you explicitly undelegate your vote.

In the first input select the account you want to delegate to and in the second input select the amount of your conviction. Remember, higher convictions means that your vote will be locked longer. So choose wisely!

The screenshot shows the "Extrinsic submission" section of the Polkadot Apps interface. It's a form for sending an extrinsic transaction. The top part shows the account "ALICE" selected as the sender. Below that, the extrinsic type is chosen as "democracy" and the function is "delegate(to, conviction)". The recipient "to" is set to "AccountId BOB". In the "conviction" field, it says "Conviction Locked2x". At the bottom, there are two buttons: "Submit Unsigned" and "Submit Transaction".

delegate

After you send the delegate transaction, you can verify it went through by navigating to the "Chain State" tab and selecting the "democracy" and "delegations" options. You will see an output similar to below, showing the addresses to which you have delegated your voting power.

The screenshot shows the Polkadot Apps Storage interface. At the top, there are tabs for 'Storage', 'Constants', and 'Raw storage'. The 'Storage' tab is selected. Below the tabs, there is a search bar with the text 'selected state query' and a dropdown menu showing 'democracy delegations(AccountId): ((AccountId,Conviction), Linkage<AccountId>)'. To the right of the search bar is a button labeled 'Get the account (and lock periods) to which anot...'. Further right is a toggle switch labeled 'include option' which is turned on, indicated by an orange circle. Below the search bar is a list item with a small circular icon containing a green and blue pattern, followed by the text 'AccountId ALICE'. To the right of this list item is a small orange 'x' button. At the bottom of the interface, there is a text box containing the query result: 'democracy.delegations: ((AccountId,Conviction), Linkage<AccountId>) [{"5FHneW46xGXgs5mUiveU4sbTyGBzmsUspZC92UhJM694ty","Locked2x"}, {"previous":null,"next":null}]'. To the right of this text box is another orange 'x' button.

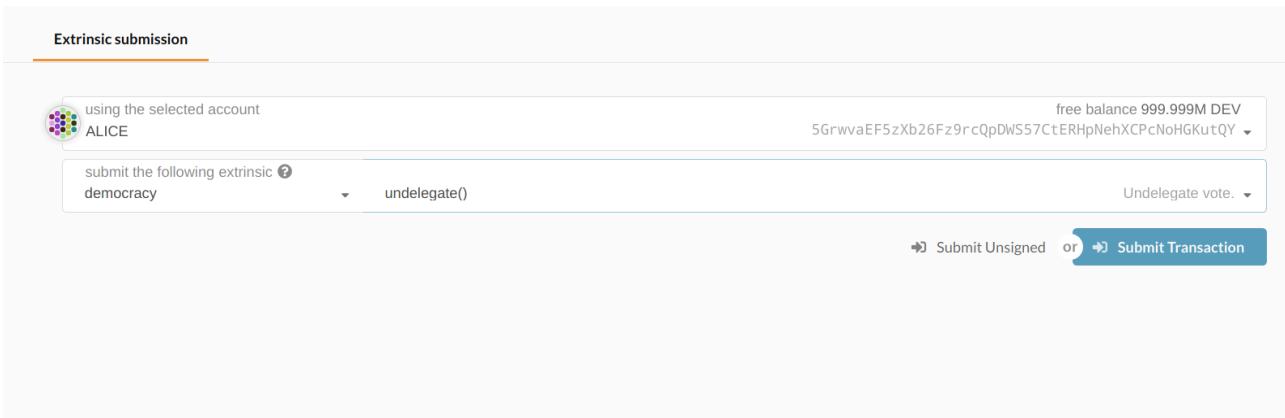
delegate state

Undelegate a Vote

You may decide at some point in the future to remove your delegation to a target account. In this case, your tokens will be locked for the maximum amount of time in accordance with the conviction you set at the beginning of the delegation. For example, if you chose "2x" delegation for four weeks lock up time, your tokens will be locked for 4 weeks after sending the `undelegate` transaction. Once your vote has been undelegated, you are in control of making votes with it once again. You can start to vote directly, or chose a different account to act as your delegate.

The `undelegate` transaction must be sent from the account that you wish to clear of its delegation. For example, if Alice has delegated her tokens to Bob, Alice would need to be the one to call the `undelegate` transaction to clear her delegation.

The easiest way to do this is from the "Extrinsics" tab of Polkadot Apps. Select the "democracy" pallet and the "undelegate" transaction type. Ensure that you are sending the transaction from the account you want to clear of delegations. Click "Submit transaction" and confirm.



undelegate

Proxies

Proxies can be used to vote on behalf of a stash account. Unlike delegation, the proxy is meant to act as a longer-term account that makes all the voting decisions for funds held in a different account. Delegation is a logical action, taken when you trust another account's judgement, while proxying is more of a recommended security practice for keeping your funds safe and using an active account with low funds instead.

Setting a proxy

Setting a proxy involves submitting a single transaction, the transaction type "setProxy" from the "democracy" pallet.

You can make this transaction from Polkadot Apps by navigating to the "Extrinsics" tab and selecting the "democracy" pallet and the "setProxy" transaction type. Send the transaction from the "Stash" account which holds the funds that you want to vote with, and the target to the proxy account that will be responsible for casting the votes going forward. In the example below, "Alice Stash" is proxying to "Alice" so that Alice can vote on behalf of Alice Stash.

Extrinsic submission

 using the selected account ALICE_STASH	free balance 10,000.000 DEV 5GNJqTPyNqANBkUVMN1LPPrxXnFouWxoe2wNSmmEoLctxiZY ▾
submit the following extrinsic ? democracy	setProxy(proxy)
 proxy: AccountId ALICE	Specify a proxy. Called by the stash. ▾ 5GrwvaEF5zXb26Fz9rcQpDW557CtERHpNehXCPcNoHGKutQY ▾
<input data-bbox="1060 444 1245 473" type="button" value="Submit Unsigned"/> or <input data-bbox="1250 444 1409 473" type="button" value="Submit Transaction"/>	

set proxy

Voting with a proxy

Making a vote on behalf of a stash requires a `proxyVote` transaction. When sending this transaction you will specify the index of the referendum that is being voted on as well as the judgement (i.e. "Aye" for approval or "Nay" for rejection).

Extrinsic submission

 using the selected account ALICE	free balance 999.999M DEV 5GrwvaEF5zXb26Fz9rcQpDW557CtERHpNehXCPcNoHGKutQY ▾
submit the following extrinsic ? democracy	proxyVote(ref_index, vote)
ref_index: Compact<ReferendumIndex> 0	Vote in a referendum on behalf of a stash. If `vote.is_aye()`, th... ▾
aye: bool Aye	▼
<input data-bbox="1060 1334 1245 1363" type="button" value="Submit Unsigned"/> or <input data-bbox="1250 1334 1409 1363" type="button" value="Submit Transaction"/>	

proxy vote

Removing a proxy

At some point you may want to remove a proxy from being able to vote on behalf of a stash account. This is possible to do by submitting a `removeProxy` transaction from the stash account, targetting the proxy account.

Extrinsic submission

using the selected account
ALICE_STASH

free balance 9.999P DEV
5GNJqTPyNqANBkUVMN1LPPrxXnFouWXoe2wNSmmEoLctxiZY ▾

submit the following extrinsic ?
democracy ▾ removeProxy(proxy)

Clear the proxy. Called by the stash. ▾

proxy: AccountId
ALICE

5GrwvaEF5zXb26Fz9rcQpDW557CtERHpNehXCPcNoHGKutQY ▾

Submit Unsigned or Submit Transaction

remove proxy

Resigning a proxy

If a proxy account wants to resign their proxy status for a different stash account this is possible to do by sending the `resignProxy` transaction. Simply call this transaction from the proxy account and all of its proxy responsibilities will be removed.



resign proxy

Democracy Features

Voting Types

Binary: a traditional yes-or-no vote • **Multi-Option:** where individuals can select multiple choices

Commit-Reveal: a scheme to commit to a specific vote by posting the hash of the vote.

At the resolution of the vote the `TallyType`, changes the method by which votes are weighed. They are enumerated below:

Tally Types

Coin-Weight: where votes that an individual account casts are weighed on a per coin—one coin is equivalent to one vote

Lock-time weight: where an account votes with a specific lock duration, with longer times corresponding to more voting power. For example, a vote may be locked for a one or two week period, with the latter time corresponding to a two-times voting power increase. Lock-time-weighting allows individual accounts with a smaller token balance to exercise more power in the governance process.

One-person-one-vote: where one account or identity can cast one vote. On Edgeware, one-person-one-vote may refer to a specific set of verified identities, such as verified Github accounts. 13 Initially, all referendum on Edgeware will be cast with both coin-weighting and lock-time-weighting. An important note, for coin-Weighted voting, a user cannot do a "partial funds vote"—users can only vote and lock up all the funds for a given account for the lock period, or not vote at all.

Implementation Delay

Following the resolution of a vote, there is a delay before implementing any change—**initially set to two weeks**.

- (i) **Parameter Note:** The implementation delay for concluded referenda is set to 2 weeks.

Delegated Voting

Additionally, the democracy module allows for delegated voting, the middle point between direct and representative democracy. At any point before a vote ends, an account can cast a vote different than how the account to whom the account may have delegated has done, essentially, **any account can overrule the decision their delegate makes**. Voluntary delegation has the potential to increase political participation, reduce strategic incentives within the election process, and ameliorate the political principal-agent problem.

Delegation can be recursive, a delegate can re-delegate to another, using their own voting power and also the delegated voting power of others. Edgeware mitigates an attack where a malicious individual may manipulate the depth of a delegation tree to an extreme depth. Instead of forcing all nodes tallying votes to traverse a deep tree of delegates, Edgeware limits the delegation to five. Cyclic delegation additionally prevented. Democracy will be extended by adding support for a different VotingType such as enabling rank-choice and anonymous voting, where accounts are directly linked to their cast ballot. Or by adding different TallyingTypes such as quadratic voting.

- (i) Delegation is limited to 5 levels deep. No vote can be delegated more than 5 times.

Identity

Judgements

After a user injects their information on chain, they can request judgement from a registrar. Users declare a maximum fee that they are willing to pay for judgement, and registrars whose fee is below that amount can provide a judgement.

When a registrar provides judgement, they can select up to six levels of confidence in their attestation:

- Unknown: The default value, no judgement made yet.
- Reasonable: The data appears reasonable, but no in-depth checks (e.g. formal KYC process) were performed.
- Known Good: The registrar has certified that the information is correct.
- Out of Date: The information used to be good, but is now out of date.
- Low Quality: The information is low quality or imprecise, but can be fixed with an update.
- Erroneous: The information is erroneous and may indicate malicious intent.

A seventh state, "fee paid", is for when a user has requested judgement and it is in progress. Information that is in this state or "erroneous" is "sticky" and cannot be modified; it can only be removed by complete removal of the identity.

Registrars gain trust by performing proper due diligence and would presumably be replaced for issuing faulty judgements.

To be judged after submitting your identity information, go to the "[Extrinsics UI](#)" and select the `identity` pallet, then `requestJudgement`. For the `reg_index` put the index of the registrar you want to be judged by, and for the `max_fee` put the maximum you're willing to pay for these confirmations.

If you don't know which registrar to pick, first check the available registrars by going to "[Chain State UI](#)" and selecting `identity.registrars()` to get the full list.

The screenshot shows the Polkadot/Substrate Portal interface with the 'Storage' tab selected. A search bar at the top contains the query: 'identity' dropdown, 'registrars(): Vec<Option<RegistrarInfo>>' input field, and a placeholder 'The s...'. Below the search bar, the result is displayed in a code block:

```
identity.registrars: Vec<Option<RegistrarInfo>>
[{"account": "FcxNWV5RESDsErjwyZmPCW6Z8Y3fbfLzmou34YZTrbcraL", "fee": 2500000000000000, "fields": []}, {"account": "Fom9M5W6Kck1hNAiE2mDcZ67auUCiNTzLBUDQy4QnxHSxdn", "fee": 5000000000000000, "fields": []}]
```

Showing all registrars

The image above reveals two registrars:

- Registrar 0, FcxNWV5RESDsErjwyZmPCW6Z8Y3fbfLzmou34YZTrbcraL charges 25 KSM per judgement
- Registrar 1, Fom9M5W6Kck1hNAiE2mDcZ67auUCiNTzLBUDQy4QnxHSxdn charges 5 KSM per judgement

To find out how to contact the registrar after the application for judgement or to learn who they are, we can check their identity by adding them to our Address Book. Their identity will be automatically loaded.

The screenshot shows the polkadot.js.org address book interface. It displays the identity of Dr. Gavin James Wood, including his legal name, email, web address, and riot handle. His KSM balance is listed as 114.246 KSM.

display	Gav	balances
legal	Dr. Gavin James Wood	114.246 KSM
email	gavin@parity.io	
web	gawwood.com	
riot	@gavofyork:matrix.parity.io	

Gav is a registrar

Gavin Wood is registrar #0.

REGISTRAR #1
1 judgement: KnownGood

display Registrar #1
legal Registrar #1
email chevdor@gmail.com
web https://www.chevdor.com
riot @chevdor:matrix.org

no tags balances ► 87.968 KSM

Chevdor is registrar #1

Chevdor is registrar #1. We pick that one.

Extrinsic submission

using the selected account
BRUNO | W3F free balance 30.246 KSM
CpjsLDC1JFyrm3ftC9Gs4QoyrkHKhZKtK7YqGTRFtTafgp ▾

submit the following extrinsic ?
identity requestJudgement(reg_index, max_fee)
Request a judgement from a registrar. ▾

reg_index: Compact<RegistrarIndex>
1

max_fee: Compact<BalanceOf>
5 KSM ▾

➡ Submit Unsigned or ➡ Submit Transaction

Requesting judgement

This will make your identity go from unjudged:

filter by name or tags

The screenshot shows a user profile for 'BRUNO | W3F'. The profile picture is a colorful hexagonal icon. The status bar at the top says 'no judgements'. Below it, a list of contact information is displayed: 'display Bruno | W3F', 'legal Bruno Škvorc', 'email bruno@web3.foundation', 'web https://bruno.id', 'twitter bitfalls', and 'riot @bruno:web3.foundation'. A small note below the list says 'An unjudged identity'.

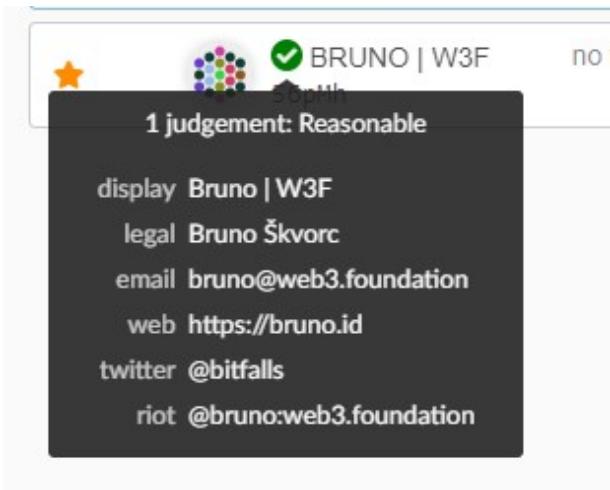
To "waiting":

The screenshot shows the same user profile for 'BRUNO | W3F'. The status bar now says 'no judgements (1 waiting)'. The rest of the contact information is identical to the previous screenshot, with the addition of '(1 waiting)' in parentheses.

A pending identity

At this point, direct contact with the registrar is required - the contact info is in their identity as shown above. Each registrar will have their own set of procedures to verify your identity and values, and only once you've satisfied their requirements will the process continue.

Once the registrar has confirmed the identity, a green checkmark should appear next to your account name with the appropriate confidence level:



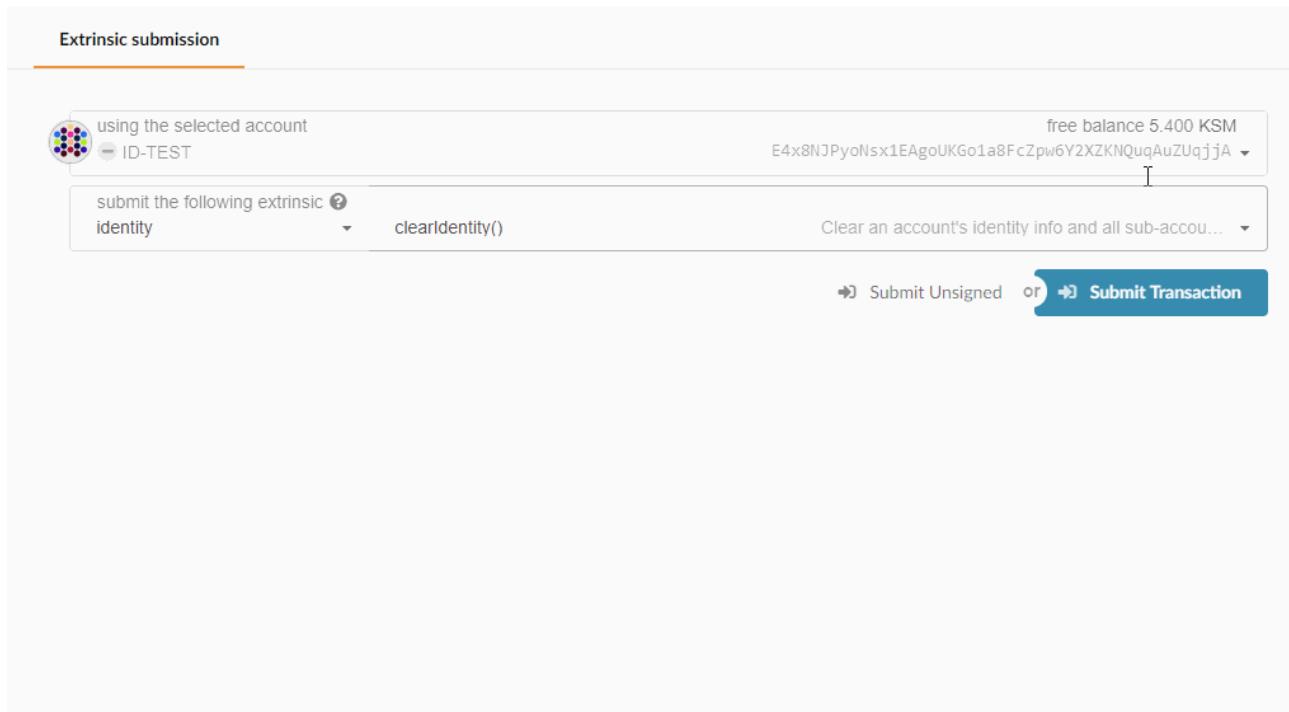
A confirmed identity

Note that changing even a single field's value after you've been verified will un-verify your account and you will need to start the judgement process anew. However, you can still change fields while the judgement is going on - it's up to the registrar to keep an eye on the changes.

Clearing and Killing

Clearing: Users can clear their identity information and have their deposit returned. Clearing an identity also clears all sub accounts and returns their deposits.

(i) Killing: The Council can kill an identity that it deems erroneous. **This results in a slash of the deposit.**



Clearing an identity

Clearing is done through the [Extrinsics UI](#).

Sub Accounts

Users can also link accounts by setting "sub accounts", each with its own identity, under a primary account. The system reserves a bond of 2 EDG for each sub account. An example of how you might use this would be a validation company running multiple validators. A single entity, "My Staking Company", could register multiple sub accounts that represent the [Stash accounts](#) of each of their validators.

(i) Parameter Note: The Sub Account Bond, the returnable fee charged for adding a sub-account, is 2 EDG per sub-account.

(i) Parameter Note: The Sub-account limit, the amount of sub-accounts an account may have, is 100.

To register a sub-account on an existing account, you must currently use the [Extrinsics UI](#). There, select the identity pallet, then `setSubs` as the function to use. Click "Add Item" for every child account you want to add to the parent sender account. The value to put into the Data field of each parent is the optional name of the sub-account. If omitted, the sub-account will inherit the parent's name and be displayed as `parent/parent` instead of `parent/child`.

The screenshot shows the "Sub account setup" section of the Extrinsics UI. At the top, it says "using the selected account" and "free balance 7.420 KSM E4x8NJPyoNsx1EAg0UKGo1a8FcZpw6Y2XZKNQuqAuZUqjjA". Below this, there's a dropdown menu for "submit the following extrinsic": "identity" and "setSubs(subs)". A note says "Set the sub-accounts of the sender." To the right are "Add item" and "Remove item" buttons. The "subs" field contains "Vec<(AccountId, Data)>". Below this, a table shows a single entry: "0: (AccountId, Data): (AccountId, Data)". The AccountId is "ID-TEST" and the Data is "None". There's also a dropdown menu for "AccountID" showing "ID-TEST". At the bottom are "Submit Unsigned" and "Submit Transaction" buttons.

Sub account setup

Note that a deposit is required for every sub-account.

Registrars

Registrars can set a fee for their services and limit their attestation to certain fields. For example, a registrar could charge 1 EDG to verify one's legal name, email, and GPG key. When a user requests judgement, they will pay this fee to the registrar who provides the judgement on those claims. Users set a maximum fee they are willing to pay and only registrars below this amount would provide judgement.

Becoming a registrar

To become a registrar, submit a pre-image and proposal into Democracy, then wait for people to vote on it. For best results, write a post about your identity and intentions beforehand, and once the proposal is in the queue ask people to second it so that it gets ahead in the referendum queue.

Here's how to submit a proposal to become a registrar:

Go to the Democracy tab, select "Submit preimage", and input the information for this motion - notably which account you're nominating to be a registrar in the `identity.setRegistrar` function.

Submit preimage

send from account  BRUNO | W3F transferrable 23.172 KSM CpjsLDC1JFyrm3ftC9Gs4QoyrkKhZKtK7YqGTRFtTa...

propose  identity addRegistrar(account) Add a registrar to the system.

account: AccountId  BRUNO | W3F/CHIRP-REGISTRAR E4x8NJPyoNsx1EAgoUKGo1a8FcZpw6Y2XZKNQuqAuZU...

preimage hash  0x90a1b2f648fc4eaff4f236b9af9ead77c89ecac953225c5fafb069d27b7131b7 imminent preimage (proposal already passed)

✖ Cancel or + Submit preimage

Setting a registrar

Copy the preimage hash. In the above image, that's

0x90a1b2f648fc4eaff4f236b9af9ead77c89ecac953225c5fafb069d27b7131b7 . Submit the preimage by signing a transaction.

Next, select "Submit Proposal" and enter the previously copied preimage hash. The `locked balance` field needs to be at least [TO ADD]. You can find out the minimum by querying the chain state under **Chain State** -> Constants -> democracy -> `minimumDeposit`.

Submit proposal

send from account  BRUNO | W3F

transferrable 23.159 KSM
CpjsLDC1JFyrm3ftC9Gs4QoyrkKhZKtK7YqGTRFtTa...

preimage hash  0x90a1b2f648fc4eaff4f236b9af9ead77c89ecac953225c5fafb069d27b7131b7

locked balance  10 KSM

 or 

Submitting a proposal

At this point, EDG holders can second the motion. With enough seconds, the motion will become a referendum which is then voted on. If it passes, users will be able to request judgement from this registrar.

Ethereum Name Services

Adding accounts to an ENS domain

This tutorial is found on the [Polkadot wiki](#) and references Kusama (KSM) and DOT for the examples. An updated version will be displayed when Edgeware is supported.

ENS (Ethereum Name Service) is a system of smart contracts on the Ethereum blockchain which allows users to claim domain names like `bruno.eth`. Supporting wallets can then allow senders to input ENS domains instead of long and unwieldy addresses. This prevents phishing, fraud, typos, and adds a layer of usability on top of the regular wallet user experience.

Note: You will need an ENS name and an Ethereum account with some ether in it to follow along with this guide. To register an ENS name, visit the [ENS App](#) or any number of subdomain registrars like [Nameth](#). Note that if you're using an older ENS name, you should make sure you're using the [new resolver](#). Visiting the ENS App will warn you about this if not. You will also need some way to use your Ethereum address - following this guide on a

personal computer is recommended. Wallets like [Frame](#) and [Metamask](#) are safe and will make interacting with the Ethereum blockchain through your browser very easy.

Despite living on the Ethereum blockchain, the ENS system has multi-chain support. In this guide you'll go through the process of adding a KSM and DOT address to ENS. We cover both KSM and DOT to show two different approaches.

Note: DOT can currently only be added using the Resolver method. KSM can be added through both methods described below.

This guide is also available in video format on [Youtube](#).

Adding via the UI

The [ENS App](#) allows an ENS domain owner to inspect all records bound to the domain, and to add new ones.

The screenshot shows a domain management interface for the domain `bruno.eth`. At the top, there are tabs for `Registrant`, `Register`, `Details` (which is selected), and `Subdomains`. Below this, a message says "Learn how to manage your name." A summary of domain details follows:

PARENT	eth	
REGISTRANT	0xB9b8EF61b7851276B0239757A039d54a23804CBb	Transfer
CONTROLLER	0xB9b8EF61b7851276B0239757A039d54a23804CBb	Set
EXPIRATION DATE	2020.10.15 at 18:58 (UTC+02:00)	Renew
RESOLVER	0x4976fb03C32e5B8cfe2b6cCB31c09Ba78EBaBa41	Set

Below this is a "RECORDS" section with a "+" button. It contains two entries:

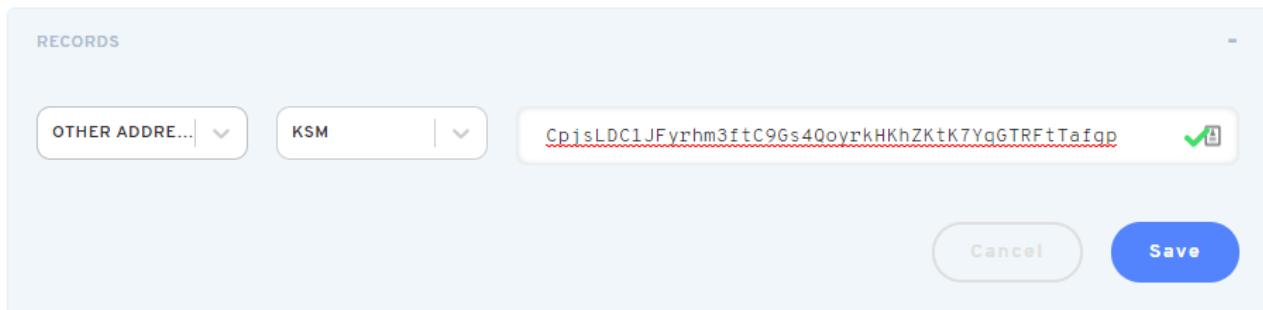
ADDRESS	0xB9b8EF61b7851276B0239757A039d54a23804CBb	edit	
Reverse record: Set to bruno.eth >			
OTHER ADDRESSES	BTC	1CbB9YvEFUbb2mXb2jZJQ9Vj9Hasg9XGz8	edit
	ETH	0xB9b8EF61b7851276B0239757A039d54a23804CBb	edit

In the example above, the domain `bruno.eth` has an Ethereum and a Bitcoin address attached. Let's attach a KSM account. First, click the `[+]` icon in the Records tab.

The screenshot shows the "RECORDS" tab with a red-bordered "+" button. Below it is an entry for an "ADDRESS".

ADDRESS	0xB9b8EF61b7851276B0239757A039d54a23804CBb	edit
---------	--	------

Then, pick "Other Addresses", "KSM", and input the Kusama address:



After clicking Save, your Ethereum wallet will ask you to confirm a transaction. Once processed, the record will show up on the domain's page:

RECORDS		
ADDRESS	0xB9b8EF61b7851276B0239757A039d54a23804CBb	
OTHER ADDRESSES	BTC	1CbB9YvEFUbb2mXb2jZJQ9Vj9Hasg9XGz8
	ETH	0xB9b8EF61b7851276B0239757A039d54a23804CBb
	KSM	CpjsLDC1JFyrhm3ftC9Gs4QoYrkKhZKtK7YqGTRFtTafgp

The same process applies to adding your DOT address.

Once the transaction is confirmed, your address will be bound to your ENS domain.

Wallet Support

There is no wallet support for ENS names for either KSM or DOT at this time, but the crypto accounting and portfolio application [Rotki](#) does support KSM ENS resolution.

Relevant links

- [ENS docs](#)
- [ENS Multi-chain announcement](#)
- [Address encoder](#)
- [Namehash calculator](#)
- [Address to pubkey converter](#)

WASM Contract Pallet

WASM Contract Pallet

WebAssembly (Wasm)

WebAssembly is used in Polkadot, Substrate and Edgeware as the compilation target for the runtime.

What is WebAssembly?

WebAssembly, shortened to simply Wasm, is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.

Why WebAssembly?

WebAssembly is a platform agnostic binary format, meaning that it will run the same instructions across whatever machine it is operating on. Blockchains need determinacy in order to have reliable state transition updates across all nodes in the peer-to-peer network without forcing every peer to run the same exact hardware. Wasm is a nice fit for reliability among the possibly diverse set of machines. Wasm is both efficient and fast. The efficiency means that it can be uploaded onto the chain as a blob of code without causing too much state bloat while keeping its ability to execute at near-native speeds.

Contracts Pallet

The Contracts pallet provides the ability for the runtime to deploy and execute WebAssembly (Wasm) smart contracts.

Wasm Engine

The Contracts pallet depends on a Wasm sandboxing interface defining the Wasm execution engine available within the runtime. This is currently implemented with wasmi, a Wasm

interpreter.

Features

The Contracts module has a number of familiar and new features for the deployment and execution of smart contracts.

Account Based

The Contracts module uses an account-based system similar to many existing smart contract platforms. To the Substrate runtime, contract accounts are just like normal user accounts; however, in addition to an AccountID and Balance that normal accounts have, a contract account also has associated contract code and some persistent contract storage.

Two Step Deployment

Deploying a contract with the Contracts module takes two steps:

Store the Wasm contract on the blockchain.

Instantiate a new account, with new storage, associated with that Wasm contract. This means that multiple contract instances, with different constructor arguments, can be initialized using the same Wasm code, reducing the amount of storage space needed by the Contracts module on your blockchain.

Runtime Environment Types

For writing contracts and interacting with the runtime, a set of types are available (e.g. AccountId, Balance, Hash, Moment). These types can be user defined for custom runtimes, or the supplied defaults can be used.

Contract Calls

Calls to contracts can alter the storage of the contract, create new contracts, and call other contracts. Because Substrate provides you with the ability to write custom runtime modules,

the Contracts module also enables you to make asynchronous calls directly to those runtime functions on behalf of the contract's account.

Sandboxed

The Contracts module is intended to be used by any user on a public network. This means that contracts only have the ability to directly modify their own storage. To provide safety to the underlying blockchain state, the Contracts module enables revertible transactions, which roll back any changes to the storage by contract calls that do not complete successfully.

Gas

Contract calls are charged a gas fee to limit the amount of computational resources a transaction can use. When forming a contract transaction, a gas limit is specified. As the contract executes, gas is incrementally used up depending on the complexity of the computation. If the gas limit is reached before the contract execution completes, the transaction fails, contract storage is reverted, and the gas fee is not returned to the user. If the contract execution completes with remaining gas, the excess is returned to the user at the end of the transaction.

The Contracts module determines the gas price, which is a conversion between the Substrate Currency and a single unit of gas. Thus, to execute a transaction, a user must have a free balance of at least $\text{gas price} * \text{gas limit}$ which can be spent.

Storage Rent

Similar to how gas limits the amount of computational resources that can be used during a transaction, storage rent limits the footprint that a contract can have on the blockchain's storage. A contract account is charged proportionally to the amount of storage its account uses. When a contract's balance goes below a defined limit, the contract's account is turned into a "tombstone" and its storage is cleaned up. A tombstone contract can be restored by providing the data that was cleaned up when it became a tombstone as well as any additional funds needed to keep the contract alive.

Contracts Module vs EVM

The Contracts module iterates on existing ideas in the smart contract ecosystem, particularly Ethereum and the EVM.

The most obvious difference between the Contracts module and the EVM is the underlying execution engine used to run smart contracts. The EVM is a good theoretical execution environment, but it is not very practical to use with modern hardware. For example, manipulation of 256 bit integers on modern architectures is significantly more complex than standard types. Even the Ethereum team has investigated the use of [Wasm](#) for the next generation of the network.

The EVM charges for storage fees only at the time of storage. This one-time cost results in some permanent amount of storage being used on the blockchain, forever, which is economically unsound. The Contracts module attempts to repair this through [storage rent](#) which ensures that any data that persists on the blockchain is appropriately charged for those resources.

The Contracts module chooses to approach contract creation using a [two-step process](#), which fundamentally changes how contracts are stored on chain. Contract addresses, their storage, and balances are now separated from the underlying contract logic. This could enable behavior like what [create2](#) provided to Ethereum or even enable repairable or upgradeable contracts on a Substrate based blockchain.

Resources

- [WebAssembly.org](#) - WebAssembly homepage that contains a link to the spec.
- [Wasmi](#) - WebAssembly interpreter written in Rust.
- [Parity Wasm](#) - WebAssembly serialization/deserialization in Rust.
- [Wasm utils](#) - Collection of Wasm utilities used in Parity and Wasm contract development.
- [Pallet-contracts](#) - The Contract module provides functionality for the runtime to deploy and execute WebAssembly smart-contracts.
- [Contracts Pallet](#)

Introduction

EVM

Edgeware has a pallet that allows developers to write EVM smart-contracts. This means that you can use Edgeware as you would with Ethereum. Edgeware is fully compatible with Ethereum's Web3 API and EVM. Here, we'll walk through a few subtle differences between Edgeware and Ethereum. Namely, Edgeware has a Proof of Stake-based consensus mechanism. This shouldn't affect you if you're building a DeFi or NFT based application. See our related documentation on [proof-of-stake](#). In the following sections we detail Edgeware<>EVM Compatiblitiy.

Full-Ethereum API and Tooling Compatibility

If you're moving some portion of your smart contracts, state, or considering porting your full set of contracts off Ethereum to Edgeware, it should 'just work'. That is the full set of your application, contracts, and tooling will remain the same. Edgeware will be able to support:

- Solidity and Serpent Based Smart Contracts
- Ecosystem Tools (e.g., block explorers, front-end development libraries, wallets—i.e Metamask)
- Development Tools (e.g., Truffle, Remix, MetaMask, ethers, web3js, truffle)
- Ethereum Tokens via Bridges (e.g., token movement, state visibility, message passing)

You can view our [tutorials](#) to get a better feel for building Ethereum smart contracts on Edgeware, and how to directly offload or migrate your Ethereum application onto Edgeware.

As previously mentioned, Edgeware is proof of stake, this does mean that smart contracts that rely on components of Ethereum's API that touch on Proof of Work–difficulty, uncles, hashrate won't work as expected on Edgeware. For those values, we have constant values set at the runtime level. Existing Ethereum contracts that rely on Proof of Work internals (e.g., mining pool contracts) will almost certainly not work as expected on Edgeware.

How Edgeware achieves Ethereum Compatibility

Edgeware achieves Ethereum compatibility in three integrated components. If you're a smart contract developer, this may just be of passing interest.

- Pallet Ethereum: which allows for full Ethereum Block Processing
- SputnikEVM: You can view the full Documentation here: <https://docs.rs/evm/>
- Pallet EVM: which allows you to deploy

EVM Balances

The "EVM" module in Substrate provides support for executing Ethereum contracts on a substrate chain. In order to perform any gas or balance-related actions on the EVM, the calling account must have a balance. How do these balances work?

Balance Conversion

In order to use Ethereum contracts on a Substrate chain, the chain must have a protocol to support the following assumptions:

1. A (32-byte) Substrate address must have a corresponding (20-byte) Ethereum address.
2. Each (20-byte) Ethereum address must have its balance maintained.

The EVM module satisfies step 1 by simply truncating the source Substrate address into an Ethereum address, taking the first 20 bytes. To satisfy step 2, the chain uses the pre-existing Substrate balances module to manage each Ethereum address, by converting Ethereum addresses back into "EVM addresses", which are 32-byte Substrate addresses. **Note that these EVM addresses have no inherent relationship to the original truncated Substrate address.**

Let us take an example:

- Consider a 32-byte Substrate address:
0x1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF .
- Truncate this to create a 20-byte Ethereum address:
0x1234567890ABCDEF1234567890ABCDEF12345678
- This Ethereum address's balance comes from the Substrate balances module's state for its corresponding EVM address, produced by hashing the above bytes with an `evm:` prefix (`0x65766D3A`). So we perform
Hash(0x65766D3A1234567890ABCDEF1234567890ABCDEF1234) =
0xAF8536395A1EEC8EDA6FB9CF36739ECF75BECF6FEA04CEEC108BBB6AA15B7CB3 , whose balance in the Balances module will be used for EVM-related operations.

- The exact hash function used is defined in the Substrate chain's runtime file. In Edgeware's case, the hashing algorithm is `Blake2`, but the `Keccak` is also a possibility.

Note that these actions are not reversible: we cannot convert from an EVM address back to its Ethereum address, nor can we convert from an Ethereum address back to its "source" Substrate address.

Managing Ethereum Balances on Substrate

Two operations are possible: we can "deposit" funds from a Substrate account into its corresponding Ethereum account, and we can "withdraw" funds from an Ethereum account back into the source Substrate account.

Deposits

Since the EVM address backing an Ethereum address is computed deterministically, as above, we can perform a standard balance transfer from our source Substrate account to the EVM address in order to seed the Ethereum account with funds.

This can be performed by calling

```
balances::transfer(prefixAndHash(truncate(account)).signAndSend(account))
```

, where
account is the source substrate account, truncate takes the first 20 bytes as the
Ethereum address, and prefixAndHash applies the evm: prefix and takes the hash as to
convert back into a 32-byte Substrate address.

Withdrawal

Since the EVM address is computed deterministically, we do not have a private key for it, so we cannot perform a balance transfer from it via normal means. As a result, the EVM module provides a special function `withdraw` for transferring funds back from an Ethereum account to the source Substrate account.

This can be performed by calling

```
evm::withdraw(truncate(account), value).signAndSend(account)
```

 where account is

the source Substrate account, and `truncate` takes the first 20 bytes as the Ethereum address.

Ethereum Balances

Ethereum balances are handled as if they were running on any Ethereum chain: gas is subtracted from the balance (the quantity of gas used can be accessed from the transaction receipt returned by the EVM module through `web3` or `truffle`), and transfers work as expected.

The balance of the 20-byte Ethereum address and the 32-byte EVM address should be identical, when compared: `web3.eth.getBalance(ethAddress)` should equal `system::balances(prefixAndHash(ethAddress)).freeBalance`, where `ethAddress` is the 20-byte Ethereum address, and `prefixAndHash` applies the `evm:` prefix and takes the hash, as explained above.

Note that the conversion rate of EVM gas to substrate tokens is defined in the Substrate runtime as `FeeCalculator`, which in Edgeware's case is set to a 1-to-1 wei-to-Substrate token mapping.

When converting, also note the `gasPrice` is specified in the EVM call, as the conversion factor from the `gasUsed` field in the receipt to wei.

(beneath is the original technical writeup)

Addresses:

- Convert a Substrate address to a EVM address: `address[0..20]` – take the first 20 bytes.
- Convert an EVM address into a Substrate address: add `evm:` prefix to data, and take the hash. **THIS IS NOT REVERSIBLE.**
 - See:
<https://gitlab.parity.io/parity/substrate/-/blob/16fdfc4a80a14a26221d17b8a1b9a95421a1576c/frame/evm/src/lib.rs#L167>

```
1 fn into_account_id(address: H160) -> AccountId32 {  
2     let mut data = [0u8; 24];
```

```

3     data[0..4].copy_from_slice(b"evm:");
4     data[4..24].copy_from_slice(&address[..]);
5     let hash = H::hash(&data);
6
7     AccountId32::from(Into::<[u8; 32]>::into(hash))
8 }
```

- Effectively: each Substrate address is mapped to a specific EVM address, which is *maintained as if it were a Substrate address* (see below). I call the (32-byte) Substrate address created from a (20-byte) EVM address the "substrate-ethereum equivalent".
 - As an example, consider the (fake) 32-byte Substrate address (it uses an `AccountId32` which is an array `[u8, 32]`, represented here as a 64 char hex string):


```
0x1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF
```
 - We first convert this to a 20-byte EVM address (represented here as a 40 char hex string): `0x1234567890ABCDEF1234567890ABCDEF1234`
 - We then convert that into a new substrate address:


```
Hash('evm:' + 0x1234567890ABCDEF1234567890ABCDEF1234) =
Hash(0x65766D3A1234567890ABCDEF1234567890ABCDEF1234)
```

 , where `Hash` is defined in the runtime as either Blake or Keccak. Edgeware uses Blake (see: <https://github.com/hicommonwealth/edgeware-node/blob/edg-frontier-up-1/node/runtime/src/lib.rs#L857>), so the final output would be, in hex:


```
0xAF8536395A1EEC8EDA6FB9CF36739ECF75BECF6FEA04CEEC108BBB6AA15B7CB3
```
 - Thus, the 32-byte "substrate account" `0x12345...` maps to the 20-byte "ethereum account" `0x12345...1234` which maps to the 32-byte "substrate-ethereum equivalent" `0xAF8536...` which functions as a valid Substrate address. **This is the Substrate address used to maintain account balances for the corresponding EVM account.**

Balances:

- An EVM address can be given a balance at genesis, or by sending balance directly to the "substrate-ethereum equivalent" as one would normally send balance between accounts.

"it seems one cannot use native substrate addresses by default to interact with the EVM, can you expand on this more? What does a user have to do if they have 0 balance at genesis"

- Each substrate address is deterministically mapped to an EVM address.
 - There is a util in `frontier/evm-address.js` which purports to perform this conversion, but doesn't seem to do it correctly based on the updated evm pallet (the util hasn't been updated in 4 months)
 - See implementations of the following utils in the frontier-tester project: <https://github.com/hicommonwealth/frontier-tester/blob/master/utils.js#L46>
 - It is doable as follows:

```

1 import { decodeAddress } from '@polkadot/util-crypto';
2
3 const convertToEvmAddress = (substrateAddress) => {
4   const addressBytes = decodeAddress(substrateAddress);
5   return '0x' + Buffer.from(addressBytes.subarray(0, 20)).toString('hex');
6 }
```

- In Rust (there is probably a simpler way to do this with refs):

```

1 fn convertToEvmAddress(address: H256) -> H160 {
2   let mut evmAddress: [u8; 20] = Default::default();
3   evmAddress[0..20].copy_from_slice(&address[0..20]);
4   evmAddress
5 }
```

- The original substrate address cannot be recovered from the EVM address alone.**
- Each EVM address is deterministically mapped to another substrate address which maintains its balance, and which can be sent funds directly via a basic transfer. This substrate address is computable as follows:

```

1 import { encodeAddress, blake2AsHex } from '@polkadot/util-crypto';
2
3 const convertToSubstrateAddress = (evmAddress, prefix = 42) => {
4   const addressBytes = Buffer.from(evmAddress.slice(2), 'hex');
5   const prefixBytes = Buffer.from('evm:');
6   const convertBytes = Uint8Array.from(Buffer.concat([prefixBytes,
7     const finalAddressHex = blake2AsHex(convertBytes, 256);
8     return encodeAddress(finalAddressHex, prefix);
```

```
9 }
```

- In Rust:

```
1 fn convertToSubstrateAddress(address: H160) -> AccountId32 {  
2     HashedAddressMapping<BlakeTwo256>::into_account_id(address)  
3 }
```

- This generated substrate address is equivalent to the EVM address for all intents and purposes, and is a deterministic computation from the original substrate address – it can be used in other pallets freely as a "proxy" for the EVM address. However, it does not have a private key, so it cannot sign.
- Funds must be withdrawn from an EVM account via the `pallet_evm::withdraw` function, as the "substrate-ethereum equivalent" does not have a known private key from which to send transactions.
 - <https://github.com/paritytech/substrate/blob/master/frame/evm/src/lib.rs#L304>
 - The caller must provide the (20-byte) EVM address that corresponds to their Substrate address – this is checked here:
<https://github.com/paritytech/substrate/blob/master/frame/evm/src/lib.rs#L148>.
 - The function converts that EVM address into the "substrate-ethereum equivalent", from which it transfers the funds.
- The `free_balance` of the "substrate-ethereum equivalent" is used as the balance of the ethereum account:
<https://github.com/paritytech/substrate/blob/master/frame/evm/src/lib.rs#L471>
- Mutations to the ethereum account's balance, such as after execution, either call `slash` or `deposit_creating` on the "substrate-ethereum equivalent" depending on whether balance was added or removed:
<https://github.com/paritytech/substrate/blob/master/frame/evm/src/lib.rs#L440>
- For a Javascript example of balances in action, see this test file in frontier-tester:
<https://github.com/hicommonwealth/frontier-tester/blob/master/web3tests/testSubstrateBalances.ts>

Gas:

- The prices of various functions in gas are defined in the `EVM_CONFIG` in the runtime.
 - Edgeware's current configuration uses a clone of the `ISTANBUL` config, with the `CreateContractLimit` raised.
 - See: <https://github.com/rust-blockchain/evm/blob/master/runtime/src/lib.rs#L245>
- The conversion rate of gas to substrate tokens is defined in the runtime as `FeeCalculator`, which in Edgeware's case is set to a 1-to-1 mapping.
- Gas is "withdrawn" from the StackExecutor before execution:
<https://github.com/paritytech/substrate/blob/master/frame/evm/src/lib.rs#L608>
- Balance changes including gas (performed by the above withdraw) are written back to the "substrate-ethereum equivalent"s balance once the EVM execution has been `apply` ed on the backend:
<https://github.com/paritytech/substrate/blob/master/frame/evm/src/backend.rs#L144>

Staking

Staking

Edgeware uses a version of proof-of-stake called Nominated Proof-of-Stake, and stakers are split into two types:

- **Validators:** Node operators who secure the network and are backed by nominators and their own stake.
- **Nominators:** EDG holders who delegate/nominate their EDG to a validator and share in the benefits.

See more details about these roles below:

→ [Intro to Roles in NPoS](#)

/edgeware-runtime/staking/intro-to-roles-in-npos

Nominate

Validate

Validation requires establishing a node, setting up your account and keys, and bonding EDG in order to verify blocks and secure the network. Follow these steps, in order, to get started.

Staking and Consensus Parameters

Reaping Threshold: The amount that an account must maintain in order to avoid deletion.

Consensus configuration

- Aura
- Runtime version: 28
- Blocktime: 6 seconds

- Session: 60 minutes (600 blocks)
- Era: 6 hours (6 sessions per era)

Staking configuration

- Validators at launch: 10
- Validator slots at launch: 60
- Bonding duration: **14 days**
- Slashes deferred duration: 7 days
- Cancellation of slashes: 3/4 of council required - need to revisit this

Intro to Roles in NPoS

Validators and Nominators

Since validator slots will be limited, most of those who wish to stake their EDG and contribute economic security to the network will be nominators. **Validators** do most of the heavy lifting: they produce new block candidates, vote and come to consensus, validate the chain's blocks.

Nominators, on the other hand, do not need to do anything once they have bonded their EDG. The experience of the nominator is similar to "set it and forget it," while the validator will be doing active service for the network by performing the critical operations.

- ⓘ As the more active participant, the validator has certain privileges regarding the payout of the staking mechanism and will be able to declare its own allocation before the share is divided to nominators.

Validators

A couple of times per day, the system elects a group of entities called **validators**, who in the next few hours will play a key role in highly sensitive protocols such as block production block finality. Their job is demanding as they need to run costly operations, ensure high communication responsiveness, and build a long-term reputation of reliability.

They also must stake their EDG, Edgeware's native token, as a guarantee of good behavior, and this stake gets slashed whenever they deviate from their protocol. In contrast, they get paid well when they play by the rules. Any node that is up to the task can publicly offer itself as a validator candidate. However, for operational reasons only a limited number of validators can be elected, expected to be hundreds or thousands.

- ⓘ **Parameter Note:** Edgeware initially allows 60 Validator slots, but this number can change through governance actions.

→ Slashing Consequences

/edgeware-runtime/staking/slashing-consequences

Nominators

The system also encourages **any** EDG holder to participate as a **nominator**. A nominator publishes a list of validator candidates that they trust, and puts down an amount of EDG at stake to support them with. If some of these candidates are elected as validators, she shares with them the payments, or the sanctions, on a per-staked-EDG basis.

Unlike validators, an *unlimited* number of parties can participate as nominators. As long as a nominator is diligent in their choice and only supports validator candidates with good security practices, the role carries low risk and provides a continuous source of revenue.

Nominating

Any potential validators can indicate their intention to be a validator candidate. Their candidacies are made public to all nominators, and a nominator in turn submits a list of any number of candidates that it supports.

There are no particular requirements for a EDG holder to become a nominator, though we expect each nominator to carefully track the performance and reputation of validators. This guide will cover how to nominate validators in the Edgeware ecosystem.

Once nominated, you will see your nominations as 'Waiting nominations' which will change to 'Active nominations' after an era(6 hours of period).

For all the nominations older than an era(6 hours), the NPoS election mechanism takes the nominators and their associated votes as input, and outputs a set of validators of the required size, that maximizes the stake backing of any validator, and that makes the stakes backing validators as evenly distributed as possible.

Based on [Sequential Phragmén election algorithm](#) your stake gets dispersed in different proportions to any subset or all of the validators you choose.

The objectives of this election mechanism are to maximize the security of the network, and achieve fair representation of the nominators. If you want to know more about how NPoS works (e.g. election, running time complexity, etc.), please read [here](#).

Nominate EDG to a Validator

See Your Nominated Amounts per Validator

- (i) Nominating a Validator means that you delegate your EDG to be used for their validation purposes, and exposes you to the risk that the Validator may 'misbehave' according to the network, resulting in theirs AND your nominated EDG

'slashed,' or taken by the system as a disincentive to misbehave. Nominate responsibly.

Step 1: Bond your tokens

Prerequisites:

- Have some EDG that you want to bond & stake in an account - we will call this the "**Stash**."
- Send only 5-10 EDG to another account that will control the nomination, we'll call this account the "**Controller**." It will be used to pay the transaction fees, *without funds to pay fees, the nomination transaction will fail.* The stash and controller are recommended to keep separate for additional security. However, one can use the same(single) account as a stash as well as a controller without compromising any functionality.
- Research and select Validators to nominate. See link below for Validator list and details.

On the [Polkadot Apps UI](#) navigate to the "Staking" option under "Network" tab.

-  Ensure you are connected to Edgeware network on the Polkadot Apps UI, instead of Polkadot or any other Substrate network. (Current network name can be found on the top-left corner.)

The "Staking Overview" subsection will show you all the active validators and their information - points (reliability), earnings, identities, etc. The "Waiting" subsection lists all pending validators that need more nominations to enter the active validator set.

The "Account Actions" subsection allows you to stake and nominate and the "Validator Stats" screen will show you some interesting in-depth graphs about a selected validator.

The "Payouts" subsection allows you to claim rewards from staking.

The "Targets" subsection will help you estimate your earnings and this is where you can also pick favorite validators and nominate selected ones conveniently.

The "Waiting" subsection lists all pending validators that are awaiting more nominations to enter the active validator set. Validators will stay in the waiting queue until they have enough EDG backing them (as allocated through the [Phragmén election mechanism](#)). It is also possible that a validator can remain in the queue for a very long time if they never get enough backing.

The "Validator Stats" subsection allows you to query a validator's stash address and see historical charts on era points, elected stake, rewards, and slashes.

Pick "Account Actions", then click the "+ Stash" option you will find on right hand side.

The screenshot shows a modal window titled "bonding preferences". It contains two sections for account selection:

- stash account:** TEST (EXTENSION) with address jWdqbKvSRMhCf7tc5pXNoXDZh2HdM9bbeZ31CcovKLY6...
- controller account:** TEST (EXTENSION) with address jWdqbKvSRMhCf7tc5pXNoXDZh2HdM9bbeZ31CcovKLY6...

A yellow warning box states: "⚠️ Distinct stash and controller accounts are recommended to ensure fund security. You will be allowed to make the transaction, but take care to not tie up all funds, only use a portion of the available funds during this period."

Below, there are fields for "value bonded" (set to 10), "balance 20,000 EDG", and a dropdown menu for "EDG". To the right, a note says: "The amount placed at-stake should not be your full available available amount to allow for transaction fees." Below this is another note: "Once bonded, it wil need to be unlocked/withdrawn and will be locked for at least the bonding duration." A "payment destination" dropdown is set to "Stash account (increase the amount at stake)". To the right, a note says: "Rewards (once paid) can be deposited to either the stash or controller, with different effects." At the bottom right are "Cancel" and "Bond" buttons.

Bonding

You will see a modal window that looks like the above.

Note: Unbonding Duration

After unbonding, there is a period of 14 days which must pass before you can make those EDGs transferable.

Select a "value bonded" that is **less** than the total amount of EDG you have, so you have some left over to pay transaction fees. Be mindful of the reaping threshold - the amount that must remain in an account lest it be burned. That amount is 0.001 in Edgeware, so it's recommended to keep at least 0.001 EDG in your account to be on the safe side.

i **Parameter Note:** The Reaping Threshold, or the minimum number of EDG to maintain an account, is currently set to 0.001 EDG (10_000_000_000_000 units.)

Choose whatever payment destination sounds good to you. If you're unsure choose "Stash account (increase amount at stake)" which will yield compounding effect.

The screenshot shows the Edgeware Staking interface. At the top, there's a navigation bar with tabs for Accounts, Network, Governance, Developer, Settings, GitHub, and Wiki. Below the navigation bar, there are several sub-tabs: Staking overview, Account actions (which is selected), Payouts, Targets, Waiting, Slashes, and Validator stats. Under the Account actions tab, there are filters: All stashes (selected), Nominators, Validators, Inactive, and buttons for Nominator, Validator, and Stash. The main content area displays a table for a stash account named 'stashes'. The table has columns for controller, rewards, and bonded amount. It shows two entries: 'TEST (EXTENSION)' and 'TEST (EXTENSION)'. Both entries have the status 'Staked' and a bonded amount of '1.0000 EDG'. There are also buttons for Session Key and Nominate. A blue banner at the bottom of the table area says 'Bonded'.

Step 2: Nominate a validator

You are now bonded. Being bonded means your tokens are locked and **could be slashed if the validators you nominate misbehave**. All bonded funds can now be distributed to up to 16 validators. Be careful about the validators you choose since you will be slashed if your validator commits an offense.

In the "Account actions" sub-tab you will find "Nominate" option corresponding to your stash(the account you've bonded) and upon clicking you will be presented with another popup asking you to select the intended validators, alternatively you can enter the validating address(es) of the validator(s) you wish to nominate.

nominate validators

The screenshot shows the 'nominate validators' screen in the Polkadot JS extension. It displays two sections: 'stash account' and 'controller account', both labeled 'TEST (EXTENSION)'. Below these are dropdown menus for 'Filter by name, address, or account index' and 'candidate accounts' (listing CN391, CouHP, CKnSn, CnsrZ, and BISON TRAILS/BISON TRAILS 7). To the right, under 'nominated accounts', are five validators listed: STAKEFISH 06, RYABINA / 6.T.ME/EDGEWARE_BOT, TSUKI/7, BISON TRAILS/BISON TRAILS 3, and HASHQUARK01. A warning message at the bottom left says: '⚠ You should trust your nominations to act competently and honest; basing your decision purely on their current profitability could lead to reduced profits or even loss of funds.' On the right, there are 'Cancel' and 'Nominate' buttons.

Nominating validators

- ⓘ Is your Nominate or Send button greyed out or not visible? Incase you are using the Polkadot JS extension, check your extension settings to ensure you have set the default network to Edgeware. Also make sure that you have both stash and controller accounts imported.

The screenshot shows the 'Staking' tab in the Polkadot JS extension. Under the 'stashes' section, it lists two stashes: TEST (EXTENSION) and TEST (EXTENSION). The first stash has a status of 'Staked' with '1,0000 EDG' bonded. To the right, a green notification bar indicates 'staking.nominate inblock' and 'system.ExtrinsicSuccess treasury.Deposit extrinsic event'. At the bottom right of the stashes table, there is a 'Stop' button.

Nominated

Select them, confirm the transaction, and you're done - you are now nominating. Your stash will start generating staking rewards within an era (6 hours). You will notice

your balance increasing whenever a validator or any nominator corresponding to it claims a payout (on behalf of every corresponding nominators).

using Subkey to Sign Transactions Securely

Coming soon.

Stop Being a Nominator (unbond)

At some point, you might decide to stop nominating one or more validators. You can always change who you're nominating, but you cannot withdraw your tokens unless you unbond them. The following guide describes how to stop nominating and then unbond to retrieve your EDGs.

On the [Polkadot Apps UI](#), **connect to the Edgeware endpoint**, navigate to the "Staking" option under "Network" tab.

On this tab click on the "Account actions" sub-tab at the top of the screen.

 **Parameter Note:** Unbonding Time: 14 days

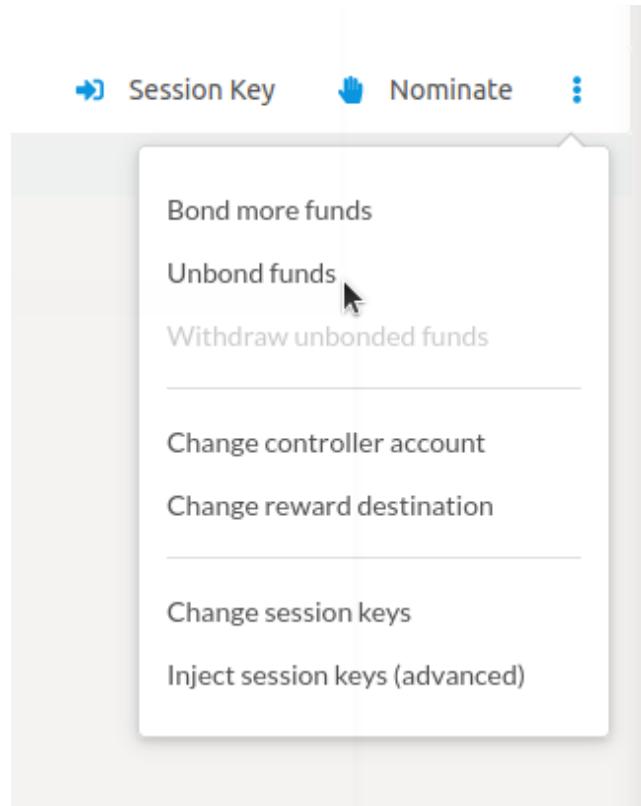
You may stop nominating at any time, but **withdrawing unbonded will fail if 14 days haven't passed since you bonded**.

Step 1: Stop Nominating

Here, click "Stop" option corresponding to your stash which will prompt you to enter sign an extrinsic via your Controller account.

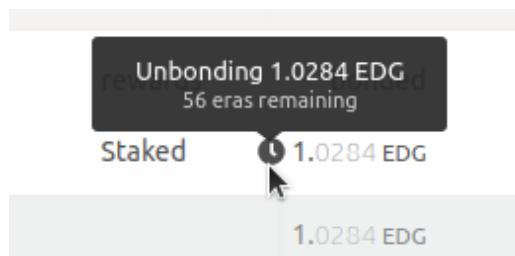
Step 2: Unbonding an amount

To unbond the amount, click the 3-dot menu corresponding to your stash from which you want to unbond EDGs and select "Unbond funds". You can then enter intended amount to unbond and select "Unbond" option which will prompt a transaction/extrinsic.



Unbonding

After you confirm this transaction, your EDGs will remain *bonded* until the unbonding period of 14 days passes. Your balance will show as "unbonding" with an indicator of how many more blocks remain until the amount is fully unlocked.



Unbonding duration

Once the 14 days of unbonding period passes, you will have to issue another(final) transaction/extrinsic: withdrawUnbonded. You can prompt this transaction/extrinsic by

simply clicking on the lock symbol corresponding to your stash in "Account actions" sub-tab.



Then, your transferrable balance will increase by the amount of tokens you've just fully unbonded.

Validating

The Validator Election Process

You can look at our quick start guide to start participating in validating

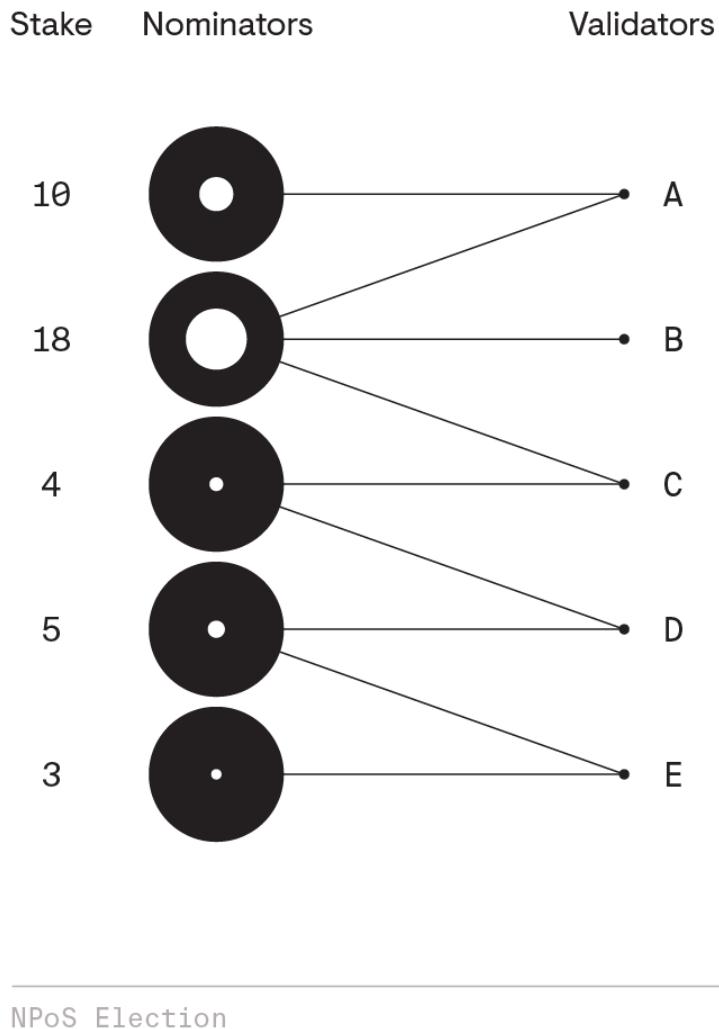
How to elect the validators, given the nominators' votes? Unlike other PoS-based projects where validators are weighted by stake, NPoS gives elected validators equal voting power in the consensus protocol.

To achieve proportional representation regardless of that fact, the total pool of their nominators' stake should be distributed among the elected validators as evenly as possible, BUT still respect the nominators' preferences. The end result, the election heuristic, should promote two key features, fair representation and security.

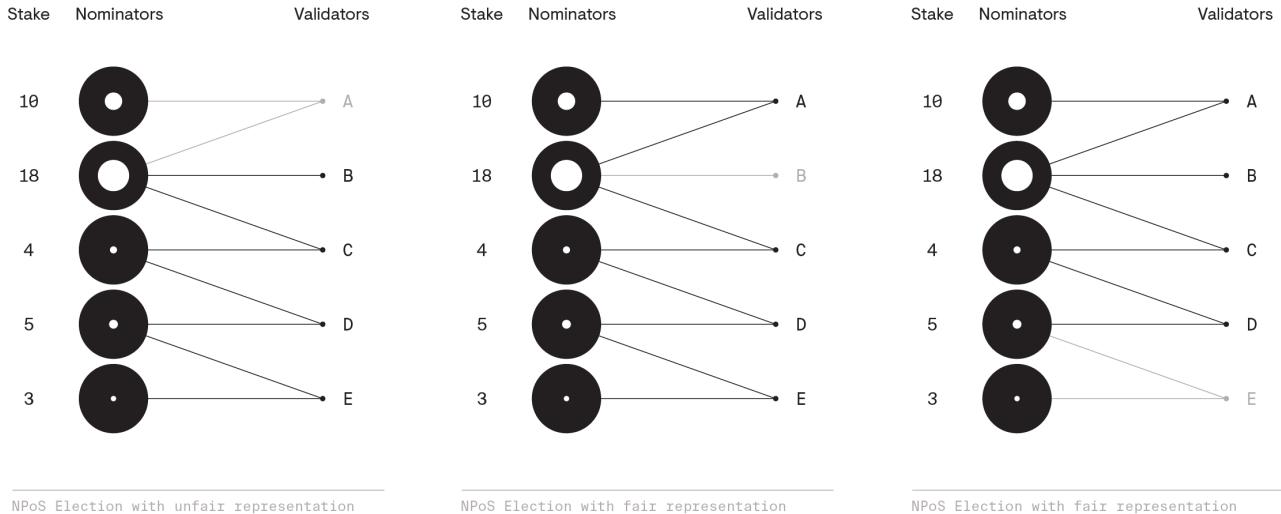
Fair Representation

In the late 19th century, Swedish mathematician Lars Edvard Phragmén proposed a method for electing members to his country's parliament. He noticed that the election methods at the time tended to give all the seats to the most popular political party; in contrast, his new method ensured that the number of seats assigned to each party were proportional to the votes given to them, so it gave more representation to minorities. The property achieved by his method is formally known as **proportional justified representation**, and is very fitting for the NPoS election because it ensures that any pool of nodes is neither over-represented nor under-represented by the elected validators, proportional to their stake. NPoS builds on top of Phragmén's suggested method and ensure this property in every election.

The illustration represents a typical input to the election process, with nominators on the left having different amounts of stake, and connected by lines to those validator candidates on the right that they trust (for simplicity, validators have no stake of their own in this example, though they will in a real scenario).



Suppose we need to elect $n=4$ validators. The fair representation property roughly translates to the rule that any nominator holding at least one n -th of the total stake is guaranteed to have at least one of their trusted validators elected. As the total stake is 40 EDG and a fourth of it is 10 EDG, the first two nominators are guaranteed to be represented by a validator. In the image below we see three possible election results: one that violates the fair representation property and two that achieve it.



Security

If a nominator gets two or more of its trusted validators elected, we need to distribute their stake among them, in such a way that the validators' backings are as balanced as possible. Recall that we want to make it as difficult as possible for an adversarial pool to get a validator elected, and they can achieve this only if they get a high enough backing. Therefore, we equate the level of security of an election result to *the minimum amount of backing of any elected validator*. For the last two election results with fair representation, we provide stake distributions which show that they achieve security levels of 6 and 9 respectively.

The election result on the right achieves a higher security level, and clearly does a better job at splitting the nominators' stake into validators' backings of roughly equal size. The goal of the NPoS election process is thus to provide a result that achieves fair representation and a security level that is as high as possible. This gives rise to a rather challenging optimization problem (it is **NP-complete**), for which we have developed fast approximate heuristics with strong guarantees on security and scalability.

To learn more about the operations side of the problem of electing validators in NPoS, view Web3 Foundation Research [technical overview](#).

Frequently Used Commands

Generate Keys with the newly added Edgeware Network ID

```
subkey -n edgeware generate
```

Running a node

To start up the Edgeware node and connect to testnet 3.0.5 (This may be out of date!, run:



The testnet shown here may be out of date. Check the version and JSON name.

```
./target/release/edgeware --chain=chains/testnet-3.0.5.json --name <INSERT_NA
```

To run a chain locally for development purposes:

```
./target/release/edgeware --chain=local --alice --validator
```

To allow apps in your browser to connect, as well as anyone else on your local network, add the `--rpc-cors=all` flag.

To allow apps in your browser to connect, as well as anyone else on your local network, add the `--rpc-cors=all` flag.

To force your local to create new blocks, even if offline, add the `--force-authoring` flag.

Generating keypairs

To create a keypair, install subkey with

```
cargo install --force --git https://github.com/paritytech/substrate subkey .
```

Then run the following:

```
subkey generate
```

To create an ED25519 keypair, run the following:

```
subkey -e generate
```

To create derived keypairs, use the mnemonic generated from a method above and run (use `-e` for ED25519):

```
subkey inspect "<mnemonic>"//<derive_path>
```

```
subkey inspect "<mnemonic>"//<derive_path>
```

For example:

```
subkey inspect "west paper guide park design weekend radar chaos space giggle
```

```
subkey inspect "west paper guide park design weekend radar chaos space giggle
```

Public nodes

Commonwealth Labs maintains several public node on mainnet and testnet. Please refer to the following page for the latest node contacts.

Syntax:

- `wss://testnet1.edgewa.re`
or (if wss: fails)
- `ws://testnet1.edgewa.re:9944`

Advanced: Staking

Staking

This guide edited from the Polkadot Wiki with gratitude

Edgeware uses NPoS (Nominated Proof-of-Stake) as its mechanism for selecting the validator set. It is designed with the roles of **validators** and **nominators**, to maximize chain security. Actors who are interested in maintaining the network can run a validator node. At genesis, Edgeware will have a limited amount of slots available for these validators, but this number will grow over time to over one thousand.

The system encourages EDG holders to participate as nominators. Nominators may back up to 16 validators as trusted validator candidates.

Validators assume the role of producing new blocks, validating blocks, and guaranteeing finality. Nominators can choose to back select validators with their stake.

The staking system pays out rewards equally to all validators. Distribution of the rewards are pro-rata to all stakers after the validator payment is deducted. In this way, the network incents the nomination of lower-staked validators to create an equally-staked validator set.

How does staking work in Edgeware?

1. Identifying which role you are

In staking, you can be either a [nominator](#) or a [validator](#).

As a nominator, you can nominate one or more (up to 16) validator candidates that you trust to help you earn rewards in EDG. You can take a look at the [nominator guide](#) to understand what you are required to do when the mainnet launches.

A validator node is required to be responsive 24/7, perform its expected duties in a timely manner, and avoid any slashable behavior.

2. Nomination period

Any potential validators can indicate their intention to be a validator candidate. Their candidacies are made public to all nominators, and a nominator in turn submits a list of any number of candidates that it supports. In the next epoch (lasting several hours), a certain number of validators having the most EDG backing get elected and become active. There are no particular requirements for a EDG holder to become a nominator, though we expect each nominator to carefully track the performance and reputation of validators.

Once the nomination period ends, the NPoS election mechanism takes the nominators and their associated votes as input, and outputs a set of validators of the required size, that maximizes the stake backing of any validator, and that makes the stakes backing validators as evenly distributed as possible. The objectives of this election mechanism are to maximize the security of the network, and achieve fair representation of the nominators. If you want to know more about how NPoS works (e.g. election, running time complexity, etc.), please read [here](#).

3. Staking Rewards Distribution

To explain how rewards are paid to validators and nominators, we need to consider **validator pools**, where a validator pool consists of an elected validator together with the nominators backing it. (Note: if a nominator n with stake s backs several elected validators, say k , the NPoS election mechanism will split its stakes into pieces s_1, s_2, \dots, s_k , so that it backs validator i with stake s_i . In that case, nominator n will be rewarded the same as if there were k nominators in different pools, each backing a single validator i with stake s_i). For each validator pool, we keep a list of nominators with the associated stakes.

The general rule for rewards across validator pools is that two validator pools get paid the **same amount of EDG** for equal work, i.e. they are NOT paid proportional to the stakes in each pool. Within a validator pool, a (configurable) part of the reward goes to pay the validator's commission fees and the remainder is paid **pro-rata** (i.e. proportional to stake) to the nominators and validator. Notice in particular that the validator is rewarded twice: once as commission fees for validating, and once for nominating itself with stake.

To estimate the inflation rate and how many EDG you can get each month as a nominator or validator, you can use this [Excel sheet](#) as a reference and play around with it by changing some parameters (e.g. validator pools, total supply, commission fees, etc.) to have a better estimate. Even though it may not be entirely accurate since staking participation is changing dynamically, it works well as an indicator.

4. Rewards Mechanism

We highlight two features of this payment scheme. The first is that since validator pools are paid the same, pools with less stake will pay more to nominators per-EDG than pools with more stake. We thus give nominators an economic incentive to gradually shift their preferences to lower staked validators that gain a sufficient amount of reputation. The reason for this is that we want the stake across validator pools to be as evenly distributed as possible, to avoid a concentration of power among a few validators. In the long term, we expect all validator pools to have similar levels of stake, with the stake being higher for higher reputation validators (meaning that a nominator that is willing to risk more by backing a validator with a low reputation will get paid more).

The following example should clarify the above. For simplicity, we have the following assumptions:

- These validators do not have a stake of their own.
- They do NOT charge any commission fees
- Reward amount is 100 EDG tokens
- The least amount of EDG to be a validator is 350

A - Validator Pool

Nominator (4)	Stake (600)	Fraction of the Total Stake	Rewards
Jin	100	0.167	16.7
Sam	50	0.083	8.3
Anson	250	0.417	41.7
Bobby	200	0.333	33.3

B - Validator Pool

Nominator (4)	Stake (400)	Fraction of the Total Stake	Rewards
Alice	100	0.25	25
Peter	100	0.25	25

John	150	0.375	37.5
Kitty	50	0.125	12.5

Both validator pools A & B have 4 nominators with the total stake 600 and 400 respectively.

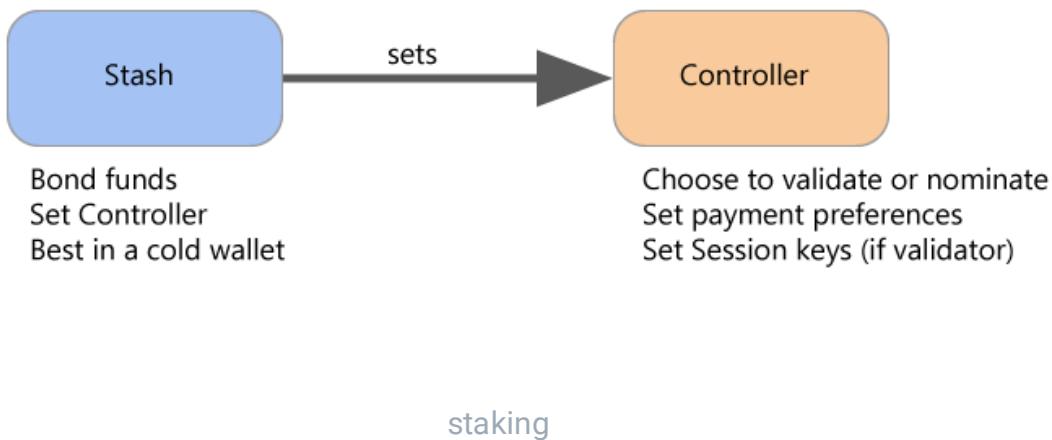
Based on the above rewards distribution, nominators in validator pool B get more rewards per EDG than those in pool A because pool A has more overall stake. Sam has staked 50 EDG in pool A, but he only gets 8.3 in return, whereas Kitty gets 12.5 with the same amount of stake.

We also remark that when the network slashes a validator slot for a misbehavior (e.g. validator offline, equivocation, etc.) the slashed amount is a fixed percentage (and NOT a fixed amount of EDG), which means that validator pools with more stake get slashed more EDG. Again, this is done to provide nominators with an economic incentive to shift their preferences and back less popular validators whom they consider to be trustworthy.

The second point to note is that each validator candidate is free to name their desired commission fee (as a percentage of rewards) to cover operational costs. Since validator pools are paid the same, pools with lower commission fees pay more to nominators than pools with higher fees. Thus, each validator can choose between increasing their fees to earn more EDG, or decreasing their fees to attract more nominators and increase their chances of being elected. We will let the market regulate itself in this regard. In the long term, we expect that all validators will need to be cost efficient to remain competitive, and that validators with higher reputation will be able to charge slightly higher commission fees (which is fair).

Accounts

There are two different accounts for managing your funds: `stash` and `controller`.



- **Stash:** This account holds funds bonded for staking, but delegates some functions to a Controller. As a result, you may actively participate with a Stash key kept in a cold wallet, meaning it stays offline all the time. You can also designate a Proxy account to vote in [governance](#) proposals.
- **Controller** This account acts on behalf of the Stash account, signalling decisions about nominating and validating. It set preferences like payout account and commission. If you are a validator, it also sets your [session keys](#). It only needs enough funds to pay transaction fees.

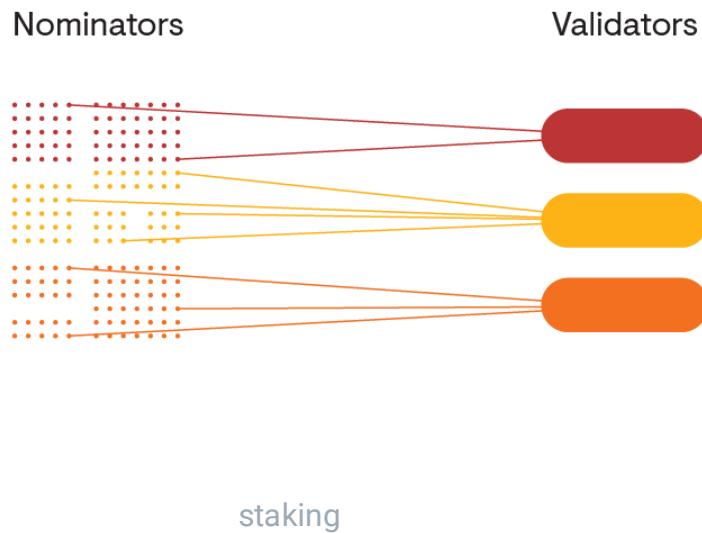
We designed this hierarchy of separate key types so that validator operators and nominators can protect themselves much better than in systems with only one key. As a rule, you lose security anytime you use one key for multiple roles, or even if you use keys related by derivation. You should never use any account key for a "hot" session key in particular.

Controller and Stash account keys can be either sr25519 or ed25519.

Validators and nominators

Since validator slots will be limited, most of those who wish to stake their DOTs and contribute economic security to the network will be nominators. Validators do most of the heavy lifting: they produce new block candidates, vote and come to consensus in GRANDPA, validate the STF of parachains, and possibly some other responsibilities regarding data availability and XCMP.

Nominators, on the other hand, do not need to do anything once they have bonded their EDG. The experience of the nominator is similar to "set it and forget it," while the validator will be doing active service for the network by performing the critical operations. For this reason, the validator has certain privileges regarding the payout of the staking mechanism and will be able to declare its own allocation before the share is divided to nominators.



Slashing

Slashing will happen if a validator misbehaves (e.g. goes offline, attacks the network, or runs modified software) in the network. They and their nominators will get slashed by losing a percentage of their bonded/staked EDG.

Validator pools with larger total stake backing them will get slashed more harshly than less popular ones, so we encourage nominators to shift their nominations to less popular validators to reduce the possible losses.

Based on Edgeware's latest codebase, the following slashing conditions have been implemented:

Unresponsiveness

For every session, validators will send an "I'm Online" message to indicate they are online. If a validator produces no blocks during an epoch and fails to send the heartbeat, it will be reported as unresponsive. Depending on the repeated offences and how many other validators were unresponsive or offline, slashing will occur.

Here is the formula for calculation:

```
1 Let x = offenders, n = total no. validators  
2  
3 min((3 * (k - (n / 10 + 1))) / n, 1) * 0.07
```

Note that if less than 10% of all validators are offline, no penalty is enacted.

Validators should have a well-architected network infrastructure to ensure the node is running to reduce the risk of being slashed. A high availability setup is desirable, preferably with backup nodes that kick in **only once the original node is verifiably offline** (to avoid double-signing and being slashed for equivocation - see below), together with proxy nodes to avoid being DDoSed when your validator node's IP address is exposed. A comprehensive guide on secure validator setup is in progress with the draft available [here](#).

GRANDPA Equivocation:

A validator signs two or more votes in the same round on different chains.

GRANDPA equivocation slashing penalty is calculated as below:

```
1 Let x = offenders, n = total no. validators  
2  
3 Min( (3 * x / n )^2, 1)
```

Validators may run their nodes on multiple machines to make sure they can still perform validation work in case one of their nodes goes down. It should be noted that if they do not have good coordination to manage signing machines, then equivocation is possible.

Notice: If a validator is reported for any one of the offences they will be removed from the validator set and they will not be paid while they are kicked out.

If you want to know more details about slashing, please look at our [research page](#).

Reward Distribution

Rewards are recorded per session and paid per era.

Example

```
1      PER_ERA * BLOCK_TIME = **Reward Distribution Time**
2
3      3600 * 6 seconds = 21,600 s = 6 hours
4
5      ***These parameters can be changed by proposing a referendum***
```

Validators can create a cut of the reward that is not shared with the nominators. This cut is a percentage of the block reward, not an absolute value. After the value gets deducted, the remaining portion is based on their staked value and split between the validator and all of the nominators who have voted for this validator.

For example, assume the block reward for a validator is 10 EDG. A validator may specify `validator_payment = 50%`, in which case the validator would receive 5 EDG. The remaining 5 EDG would then be split between the validator and their nominators based on the proportion of stake each nominator had. Note that validators can put up their own stake, and for this calculation, their stake acts just as if they were another nominator.

Rewards can be directed to the same account (controller) or to the stash account (and either increasing the staked value or not increasing the staked value). It is also possible to top-up / withdraw some bonded EDG without having to un-stake everything.

FAQs

Why stake?

- 10% inflation/year when the network launches
- 50% targeted active staking
- ~20% annual return

Why not stake?

- Tokens will be locked for 28 hours after unbonding. (See Parameters page for most recent updated numbers)
- Tokens can be slashed as punishment if the validator is not running properly.

How many validators will Edgeware have?

The plan is to start with 60 open validator positions and open more gradually. The top bound on the number of validators has not been determined yet.

Resources

- [How Nominated Proof of Stake will work in Polkadot](#) - Blog post by Web3 Foundation researcher Alfonso Cevallos covering NPoS in Polkadot.
- [Secure validator setup](#)

The Sequential Phragmen Method

What is the sequential Phragmen method?

The sequential Phragmen method is a multi-winner election method introduced by Edvard Phragmen in the 1890s.

The quote below taken from the reference [Phragmen paper](#) sums up the purpose of the sequential Phragmen method:

The problem that Phragmen's methods try to solve is that of electing a set of a given numbers of persons from a larger set of candidates. Phragmen discussed this in the context of a parliamentary election in a multi-member constituency; the same problem can, of course, also occur in local elections, but also in many other situations such as electing a board or a committee in an organization.

Where is the Phragmen method used in Edgeware?

NPoS: Validator Elections

The sequential Phragmen method is used in the Nominated Proof-of-Stake scheme to elect validators based on their own self-stake and the stake that is voted to them from nominators. It also tries to equalize the weights between the validators after each election round. Since validators are paid equally in Polkadot, it is important that the stake behind each validator is spread out. The equalization method is ran twice for every validator election. The first iteration will do a rough equalization among all validator candidates in order to determine the subset that will become the next active validators. The second iteration runs only among the elected candidates to equalize the stake between the ones which are elected.

Council Elections

The Phragmen method is also used in the council election mechanism. When you vote for council members, you can select up to 16 different candidates, and then place a reserved

bond which is the weight of your vote. Phragmen will run once on every election to determine the top candidates to assume council positions and then again amongst the top candidates to equalize the weight of the votes behind them as much as possible.

What does it mean for node operators?

Phragmen is something that will run in the background and requires no extra effort from you. However, it is good to understand how it works since it means that not all the stake you've been nominated will end up on your validator after an election. Nominators are likely to nominate a few different validators that they trust will do a good job operating their nodes.

You can use the [offline-phragmen](#) script for predicting the outcome of a validator election ahead of a new era beginning.

External Resources

- [W3F Research Page on Sequential Phragmen Method](#) - The formal adaptation of the Phragmen method as applied to Polkadot validators.
- [Python Reference Implementations](#) - Implementations of Simple and Complicated Phragmen methods.
- [Substrate Implementation](#) - Rust implementation used in the Substrate Runtime Module Library.
- [Phragmen's and Thiele's Election Methods](#) - 95-page paper explaining Phragmen's election methods in detail.
- [Offline Phragmen](#) - Script to generate the Phragmen validator election outcome before the start of an era.

Slashing Consequences

Slashing will happen if a validator misbehaves (e.g. goes offline, attacks the network, or runs modified software) in the network. They and their nominators will get slashed by losing a percentage of their bonded/staked EDG.

- (i) Validator pools with larger total stake backing them will get slashed more harshly than less popular ones, so we encourage nominators to shift their nominations to less popular validators to reduce the possible losses.

Based on Substate's latest codebase, the following slashing conditions have been implemented:

Unresponsiveness

For every session, validators will send an "I'm Online" message to indicate they are online. If a validator produces no blocks during an epoch and fails to send the heartbeat, it will be reported as unresponsive. Depending on the repeated offenses and how many other validators were unresponsive or offline, slashing will occur.

Here is the formula for calculation:

```
1 Let x = offenders, n = total no. validators  
2  
3 min((3 * (k - (n / 10 + 1))) / n, 1) * 0.07
```

Note that if less than 10% of all validators are offline, no penalty is enacted.

Validators should have a well-architected network infrastructure to ensure the node is running to reduce the risk of being slashed. A high availability setup is desirable, preferably with backup nodes that kick in **only once the original node is verifiably offline** (to avoid double-signing and being slashed for equivocation - see below), together with proxy nodes to

avoid being DDoSed when your validator node's IP address is exposed. A comprehensive guide on secure validator setup is in progress with the draft available [here](#).

GRANDPA Equivocation

A validator signs two or more votes in the same round on different chains.

BABE Equivocation

A validator produces two or more blocks on the relay chain in the same time slot.

GRANDPA and BABE equivocation slashing penalty is calculated as below:

```
1 Let x = offenders, n = total no. validators  
2  
3 Min( (3 * x / n )^2, 1)
```

Validators may run their nodes on multiple machines to make sure they can still perform validation work in case one of their nodes goes down. It should be noted that if they do not have good coordination to manage signing machines, then equivocation is possible.

Notice: If a validator is reported for any one of the offences they will be removed from the validator set and they will not be paid while they are kicked out.

If you want to know more details about slashing, please look at our [research page](#).

Reward Distribution

Note that Kusama runs approximately 4x as fast as Polkadot, except for block production times. Polkadot will also produce blocks at approximately six second intervals.

Rewards are recorded per session — approximately one hour on Kusama and four hours on Polkadot — and paid per era. It takes approximately six hours to finish one era, twenty-four

hours on Polkadot. Thus, rewards will be distributed to the validators and nominators four times per day on Kusama and once per day on Polkadot.

Example

```
1     PER_ERA * BLOCK_TIME = **Reward Distribution Time**  
2  
3     3600 * 6 seconds = 21,600 s = 6 hours  
4  
5     ***These parameters can be changed by proposing a referendum***
```

Validators can create a cut of the reward that is not shared with the nominators. This cut is a percentage of the block reward, not an absolute value. After the value gets deducted, the remaining portion is based on their staked value and split between the validator and all of the nominators who have voted for this validator.

For example, assume the block reward for a validator is 10 DOTs. A validator may specify `validator_payment = 50%`, in which case the validator would receive 5 DOTs. The remaining 5 DOTs would then be split between the validator and their nominators based on the proportion of stake each nominator had. Note that validators can put up their own stake, and for this calculation, their stake acts just as if they were another nominator.

Rewards can be directed to the same account (controller) or to the stash account (and either increasing the staked value or not increasing the staked value). It is also possible to top-up / withdraw some bonded DOTs without having to un-stake everything.

Inflation

Treasury

Treasury

Edgeware uses a treasury mechanism to achieve self-sustainability and permit creative incentivization of all stakeholders in the ecosystem, including core and dapp development, ecosystem support and much more.

These funds can be deployed through a Treasury Spend action which is subject to council approval but can be proposed by any EDG holder.

Proposing a Treasury Spend

When a stakeholder wishes to propose a spend from the treasury, **they must reserve a deposit totaling 5% of the proposed spend, with a minimum of 1000 EDG**



Parameter Note: The Treasury Spend Deposit is 5% of the proposed spend action, with a minimum of 1000 EDG.

This deposit will be slashed if the proposal is rejected, and returned if the proposal was accepted.

Plausible Project Concepts

Proposals may consist of (but are not limited to):

- Infrastructure deployment and continued operation.
- Network security operations (monitoring services, continuous auditing).
- Ecosystem provisions (collaborations with friendly chains).
- Marketing activities (advertising, paid features, collaborations).
- Community events and outreach (meetups, pizza parties, hackerspaces).
- Software development (wallets and wallet integration, clients and client upgrades).

- Core Technology: Implementing scaling technologies such as runtime improvements, sharding, and off-chain extensions.
- Governance: Governance systems, including on-chain identity as well as tools for organizing and coordinating the work of core developers.
- Developer experience: A mature toolchain for developing, debugging, and testing smart contracts.
- User experience: Wallets and user experience primitives (e.g., JavaScript libraries) to make decentralized apps simple and easy to use.
- Ecosystem support: Engaging developers, end users, and other stakeholders through in-person events.

Projects that serve the broader Edgeware ecosystem are much more likely to succeed than other kinds of efforts. This list is not exhaustive, but it gives you a sense of what sorts of things are good propositions.

The treasury is ultimately controlled by the stakeholders, and how the funds will be spent is up to their judgment.

EDG Accrual to the Treasury

The treasury obtains funds in several ways that mimic governments - minting, fees and taxes.

1. **Minting:** A portion of the EDG produced with each block goes to the treasury.
2. **Slashing:** When a validator is slashed for any reason, the slashed amount is sent to the Treasury. Slashed EDG may also accrue to the treasury through failed governance proposals.
3. **Transaction fees:** A portion of each block's transaction fees goes to the Treasury, with the remainder going to the block author.
4. **Staking inefficiency:** **Inflation** is designed to be ~20% in the first year, and the ideal staking ratio is set at 80%, meaning 80% of all tokens should be locked in staking. Any deviation from this ratio will cause a proportional amount of the inflation to go to the Treasury. In other words, if 80% of all tokens are staked, then 100% of the inflation goes to the validators as reward. If the staking rate is greater than or less than 80%, then the validators will receive less, with the remainder going to the Treasury.
5. **Lost Deposits:** These may be abandoned bonds from voting, proposals or otherwise.

 **Parameter Note:** The *ideal staking ratio* used in the Edgeware economic model is currently 80%. *Inflation* is currently set to about 20%, or 158 EDG per block.

Conducting a Treasury Spend Proposal

Once you have an idea that might be appropriate for funding from the Edgeware Treasury, you can complete several steps to test the public interest, convey your argument, and then submit your formal treasury proposal. This guide is written as a recommendation both to proposers and decision-makers, and will track best practices over time as they emerge.

Similar to [traditional RFPs](#) (request for proposals) that governments use to solicit bids for work, treasury spends optimize the productive value of the EDG and seek to extract maximum work for their recipients. Proposers should be prepared to defend their position or experience competitive bids for the same or similar work.

Technical Process of a Treasury Spend

1. Pre-Proposal Stage - Planning, writing, advocacy, outreach.
2. Treasury Spend Proposed
3. Council either adopts or refuses the proposal.
4. If approved, it enters a queue within a budget period of 24 days.
 1. The treasury attempts to fulfill as many spends in the queue within this time period.
 2. At the end of a budget period, the remaining treasury balance is subject to a small percentage burn. This incentivizes the usage of funds and creates deflation through the destruction of EDG.
5. If refused, the treasury proposal remains able to be accepted until the end of the 24 day period, then it is removed from the consideration table but can be resubmitted.

Proposal Description Document

The first step is to describe the proposal in full to the community. A great place to start is a single document that contains all the following information. Once the document is ready, create the on-chain spend in the next step, then share the document after.

1. Date of Proposal
2. Who is proposing?
 1. What is their motivation?
 2. Do they have any conflicts of interest?
 3. To who do these payments go?
 1. For building, who will be handling what?
 2. What is their background and expertise? Describe the whole team.
3. What is the historical context of this proposal?
 1. What previous actions does this proposal relate to?
 2. Does it have any impacts on future actions to note?
4. Clear, short Problem Statement and Proposed Solution
 1. Evidence for that Solution
 2. Who does this solution help?
5. Describe, if relevant, the Milestone Structure of goals and disbursements.
6. Are these funds going to be used for non-funding, on-chain purposes? e.g Will you be using these funds to vote, nominate or bond for staking?
7. Financial Details for all Proposed Transfers
 1. Amount/s requested
 2. Financial timeline - when are installments due?
 3. What are the Addresses of the fund recipients?
 4. Who will be managing the funds, how can we contact them?
 5. How will the manager of the funds report on status proactively? Where we can follow progress and ask questions?
8. License (if applicable)

Propose The Spend On-Chain

To propose a treasury spend, **a deposit totaling 5% of the proposed spend amount, with a minimum of 1000 EDG is required. This deposit will be slashed if the proposal is rejected, and returned if the proposal was accepted.**

One way to create the proposal is to use the Polkadot JS Apps [website](#). From the website, use either the [extrinsics tab](#) and select the Treasury pallet, then `proposeSpend` and enter the desired amount and recipient, **or** use the [Treasury tab](#) and its dedicated Submit Proposal button:

Submit treasury proposal

submit with account [?](#)
 BRUNO | W3F CpjsLDC1JFyrm3ftC9Gs4QoyrkHKhZKtK7Y... ▾

beneficiary [?](#)
 BRUNO | W3F CpjsLDC1JFyrm3ftC9Gs4QoyrkHKhZKtK7Y... ▾

value [?](#)
100 KSM ▾

 Cancel   Submit proposal

The system will automatically take the required deposit, picking the **higher** of the following two values: 1000 EDG or 5% of the requested amount. If the user cannot pay the deposit, an error will be returned and the proposal creation will fail.

 Treasury overview 

proposals	approved	available	spend period
0	1	114.194k	75,238/86,400

 Submit proposal

proposals

11	 BRUNO W3F 56pMh	bond 20.000 KSM	beneficiary  BRUNO W3F	value 100.000 KSM	 Send to council
----	--	-----------------------	--	-------------------------	--

approved

No approved proposals

A proposal ready for the council to consider

Once created, your proposal will become visible in the Treasury screen and the council can start voting on it.

Note the on-chain # of your treasury proposal- shown as a large number on the left of the row (11 in the image above), you'll want it for the next step.

Now the council can take action, either turning it into a motion to approve or a motion to reject. 51% or more of the council must agree to take an action on any treasury spend.

(i) Parameter Note: The *Treasury Spend Deposit* is 5% of the proposed spend action, with a minimum of 1000 EDG. *Treasury Spends* require 51% of the council to agree.

Speak with the Community

Because the formal spend proposal lacks metadata for efficiency, sharing the details about the proposal off-chain is essential. We recommend using [Commonwealth.im/Edgeware](#) and the community channels including telegram and discord.

- Post a discussion thread on Commonwealth
- Tag the title and body with **EDG_TP_#** where # is the official number of your treasury proposal, found on block explorers or Polkadot UI.
- Announce the thread in other channels like Telegram to encourage awareness and debate.

Tagging helps users understand what module is being activated and what threads connect to what actions. Expect questions, challenges, and other remarks. This can begin before or after you create your formal treasury proposal on-chain, but you won't know the TP# until then.

Tipping Function

(i) Substrate Documentation for this function is at:

https://substrate.dev/rustdocs/master/pallet_treasury/index.html#tipping

The treasury module also contains a more ad-hoc way of distributing funds from the treasury to a single recipient, called Tipping.

1. Users join a tipping round.
2. Each user names an amount of EDG that a beneficiary may deserve, subjectively.
3. Once 50% of these users submits their amount, a countdown begins for the remaining undecided.
4. Once the countdown ends, the tipping round closes.
5. The Treasury calculates the median of the tip submissions.
6. The beneficiary receives that median amount of EDG from the Treasury.

Function Supported by UI at:

Polkadot Apps

Lockdrop

The Edgeware lockdrop was a token distribution event that occurred between June 1-Aug 30 2019. **It is now closed.** Please visit our [blog](#) for more information.

- ⓘ The EDG tokens will be created upon Network Launch - currently scheduled for 10am, Monday, Feb 17, 2020. All EDG created through the lockdrop will exist at their owner's wallet addresses automatically. See the Get Started page for managing your new EDG and Edgeware account, below. There is no need to 'retrieve' your EDG.

Guides

- [Check the Status of Your Lock Duration and Unlock Date](#)

/edgeware-runtime/lockdrop/check-the-status-of-your-lock-duration-and-unlock-date

- [Retrieve your Locked ETH](#)

/quickstart/retrieve-your-eth

Description

The Edgeware network was launched with a 'lockdrop' of Edgeware tokens to Ether holders.

A Lockdrop is an airdrop where token holders of one network lock 5 their tokens in a smart contract for a fixed or variable time period, the 'lock duration.' The purpose of this novel method of token distribution is to more fairly, securely and widely align incentives among the active participants, select for and incentivize committed initial participants of the network,

and leverage existing token distributions in order to create relatively more diverse and wide distributions.

In the case of Edgeware, Ether holders will be able to participate through two interaction types: a '**Lock**' where they send and make inaccessible their ETH tokens for an elected duration of 3, 6 or 12 months, or to '**Signal**' their intent to participate in the Edgeware network.'Signaling,' is similar to an airdrop in that no lock duration is imposed and the participant's tokens remain accessible however Signal participants receive a reduced allocation compared to a Lock interaction.

For those that choose to Lock their tokens, longer lock duration corresponds to proportionally more Edgeware tokens received, and these are distributed at the launch of the network. The Lockdrop process is a more fair and secure way to distribute tokens in a Proof-of-Stake network for the following reasons:

- **A non-zero opportunity cost:**

To participate in the lockdrop, individuals are forgoing the opportunity cost of ETH for the duration of the lockdrop—e.g. being able to lend on Compound. For long-term ETH holders, this opportunity cost is irrelevant. For this reason, long-term ETH holders are most aligned with the EDG lockdrop. The minting of EDG tokens will allow token holders to participate in all the rights allocated to Edgeware participants—becoming a validator or voting on network proposals.

- **Downside protection:**

At the end of the lockdrop, participants will have access to two productive utility tokens, ETH and EDG respectively. As Edgeware is a new and experimental chain, in the event of a malicious attack or exploit on the Edgeware network, lockdrop participants will still retain their ETH, eliminating the long-term risk of participation.

- **Easy, open and accessible process:**

A single transaction that sends ETH to the Edgeware Master Lockdrop Contract allows one to receive EDG tokens. Any account can perform this from a hardware or software wallet (e.g., Trezor, Metamask, and more). Moreover, any ETH holder can participate.

- **Economic security:**

Bootstrapping security on a new PoS chain. Ethereum is bootstrapping the Serenity release from an initial crowd-sale and PoW block reward. In the same way, Edgeware can use the already-wide distribution of Ethereum holders to bootstrap the economic security of the Edgeware chain.

- **Contract security:**

When calling lock from the Master Lockdrop Contract, an individual Lockdrop User Contract is created, which actually holds the time-locked ETH of a participant. This significantly improves fund security, because value is not stored

Lockdrop Allocation Formula

Concluding Parameters

From the [Lockdrop Conclusion Metrics page](#), backed up here:



The final ETH/EDG ratio is **1 ETH : 1,156 EDG**.

Parameter	Value
Total Locked ETH	1,199,728 ETH
Total Signalled ETH	4,346,544 ETH
EDG Distributed to Lockers	64.1%
EDG Distributed to Signals	25.9
EDG Distributed via Genesis	10%
Effective Lockers ETH	2772238 ETH
Number of ETH Addresses participating in Locks	2869
Number of ETH Addresses Participating in Signals	1922
Effective Signalers ETH	1120407 ETH
Network Launch Date	Feb 17 2020
ETH in Generalized Locks	313,872 ETH
Additional effective ETH attributable to generalized lock	251,098 ETH



Effective ETH is contributed ETH modified by the bonus, lock weight and EDG/ETH ratio.

Lock-Allocation Calculation

In the lockdrop event, the number of EDG actually obtained by each locking user is determined by a formula:

$$timingBonusModifer * lockWeight * userETHLocked * (totalEDG/totalETH)$$

Variables

Timing Bonus Ratio: User-Controlled Parameter.

The earlier you lock in the event schedule's 7 bonus periods, the higher this bonus modifier is.

Per the above schedule, the parameter options are:

Time Period	Bonus Modifer
June 1 - June 15	1.5
June 16- June 30	1.35
July 1 - July 15	1.23
July 16 - July 30	1.14
July 31 - August 14	1.08
August 15- August 29	1.05
August 30 - August 31	1

Weight: User-Controlled Parameter.

Participating via a three-month lock will get 1(weight) when distributing EDG tokens;

Participating via a twelve-month lock will get 2.2 (weight) when distributing EDG tokens.

User's Locked ETH: User-Controlled Parameter

The specific number of ETH the user locks.

Total Allocatable EDG: System Parameter

**_Total EDG in at Edgeware genesis is 5 billion, or 5,000,000,000. 4.5 billion is distributed via the lockdrop event. Therefore, total allocatable EDG is 4,500,000,000.

Total Locked ETH: Aggregated User Contribution

The total number of ETH locked.

Ratio of EDG to ETH:

The number of lockdropped EDGs that one ETH can obtain. This parameter cannot be known until the end of the lockdrop event because the total ETH locked must be known. Since **the event closed and stats have been published**, it is understood to be **1 ETH : 1,156 EDG**.

Find your Lockdrop User Contract (LUC)

There are two ways to discover your LUC address if you haven't stored it:

Block Explorer

Visit a block explorer like [Etherscan.io](#) or others.

Enter your participating ETH address and view the list of transactions to find one or more that show the destination of "Edgeware Lockdrop Contract" It may also say "Old" - this is still a valid lock.

Your block explorer may not give the contract title. In this case, transactions to the following contract addresses are to the Master Lockdrop Contracts:

- 0x1b75B90e60070d37CfA9d87AFFD124bB345bf70a (V.1, Old)
- 0xFEC6F679e32D45E22736aD09dFdF6E3368704e31 (v.2 New)

Click the transaction ID of these to see the final destination of the funds - the MLC does not hold your ETH, but sends it immediately to a newly created LUC. The transaction should show you a trail from your ETH Address -> MLC -> LUC.

Overview	Internal Transactions	Event Logs (1)	State Changes	Comments
② Transaction Hash:	0x2575399ca9e0d36f1e9e16e8c8b0f449455fdebb7f4601e4b27ab4e7c50cd269 🔗			
② Status:	✓ Success			
② Block:	8452005	1018479 Block Confirmations		
② Timestamp:	① 166 days 6 hrs ago (Aug-30-2019 02:12:20 PM +UTC)			
② From:	0x544053def7421d905f07f3859cc011e4b78b6ce2 🔗			
② To:	② Contract 0x1b75b90e60070d37cfa9d87affd124bb345bf70a (Edgeware: Old Lockdrop) ✓ 🔗 └ TRANSFER 493.5 Ether From 0x1b75b90e60070d37cfa... To → 0x5cb21b128a9b0df22f2...			
② Value:	493.5 Ether	(\$134,764.98)		
② Transaction Fee:	0.000494528 Ether (\$0.14)			
Click to see More ↴				
② Private Note:	To access the Private Note feature, you must be Logged In			

LUC Address 

See this example using [Etherscan.io](#)

Click into your LUC addresses to check the LUC for balance and contract data.

Lockdrop Stats Tool

Visit the lockdrop stats page on Commonwealth.im:

<https://commonwealth.im/edgeware/stats>

If this page is unavailable or not functional, switch to the Block Explorer method tab.

Scroll to the bottom and enter your participating ETH address/es into the bottom field.

This will show you:

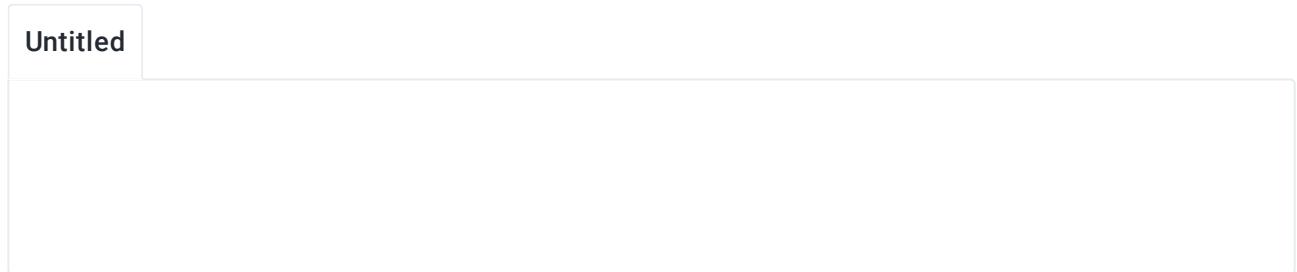
- All instances of your LUCs generated from that address
- Their unlock times
- And ETH locked per LUC.

→ [Retrieve your Locked ETH](#)

/quickstart/retrieve-your-eth

Check the Status of Your Lock Duration and Unlock Date

There are two ways to check on your lock duration. The easiest way is through the Commonwealth Unlock or Lockdrop Stats tool, the other way requires that you look at the lock-initiating transaction on a block explorer.



Visit either

- [Commonwealth's Lockdrop Stats Tool](#)
- [Commonwealth's Unlock Tool](#) (Can also automate unlocking of ETH)

Enter your participating ETH address to view all lockdrop user contracts associated with the address.

Prerequisites

- The ETH address you locked from
 - The transaction that you sent to trigger the lockdrop entry.
-

Steps

Find the lock transaction in [a block explorer like Etherscan.io](#), note the date of the send.

In the transaction details, find the Input Data (see screenshot below,) and find the segment that shows [0]:0000000000 ... and note the number at the end of the segment.

If that number is ...

θ = A three month lock duration

1 = A six month lock duration

2 = A twelve month lock duration

Add the month value to the transaction date to find your unlock date and see if that date has passed yet. If so, your LUC is ready to release your ETH.

→ Retrieve your Locked ETH

/quickstart/retrieve-your-eth

Example

For instance, in the screenshot above, the date is `June-15-2019`, and the lock duration is `2`, meaning 12 months. June 15, 2019 plus 12 months is a year later, or `~June 15, 2020`.

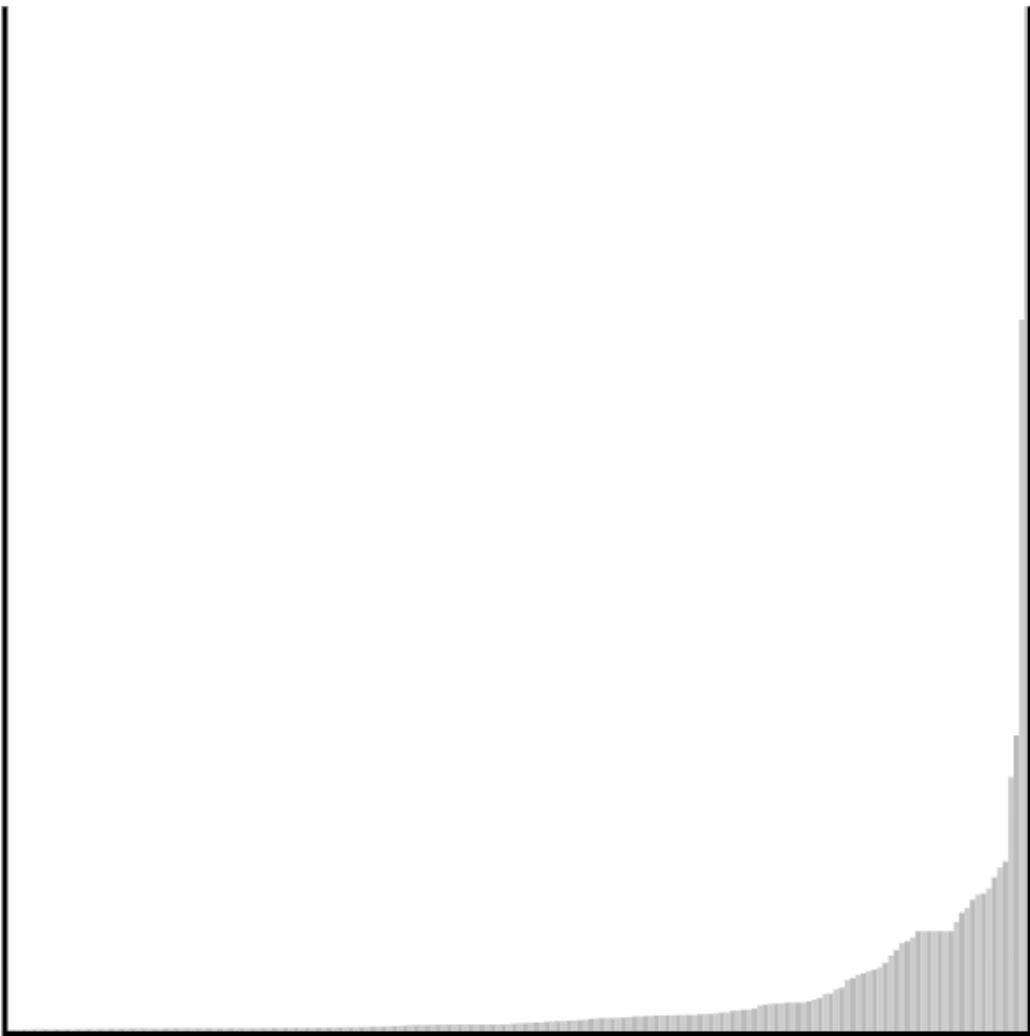
Gini Coefficient of Edgeware

Gini Coefficient Analysis

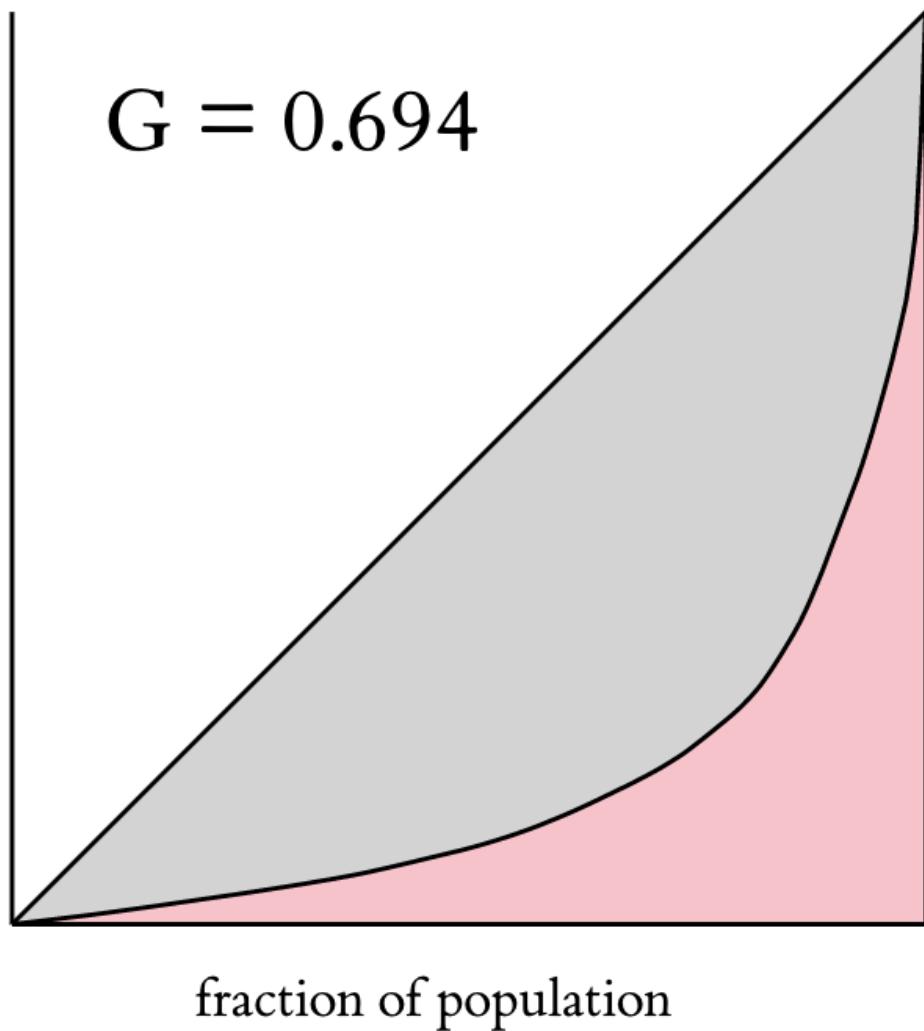
The total ETH locked in the standard process finalized at 1,199,728 ETH, with signaled funds at 4,346,544 ETH. This demonstrates strong confidence and interest in both the contracts underlying the lockdrop, but also the advance in blockchain technology that Edgeware represents. Today there exist over 4000 different Edgeware addresses largely created in the lockdrop process, from over 10,000 transactions.

The lockdrop mechanism was primarily designed to distribute a utility token in a more decentralized and equal fashion while strongly selecting for committed users and holders, this was achieved by ‘supercharging’ the power of small amounts by allowing for the use of time as leverage: lock longer, get more.

For many cryptonetworks, the decentralization of the system is analyzed using [the gini coefficient](#), which looks at the distribution of wealth. We provide the gini coefficient of 0.694 for Edgeware. The following charts show the distribution of Edgeware tokens, and then the [Lorenz curve](#) and gini coefficient of our analysis.



Distribution of Edgeware tokens of top 4.2% of Accounts

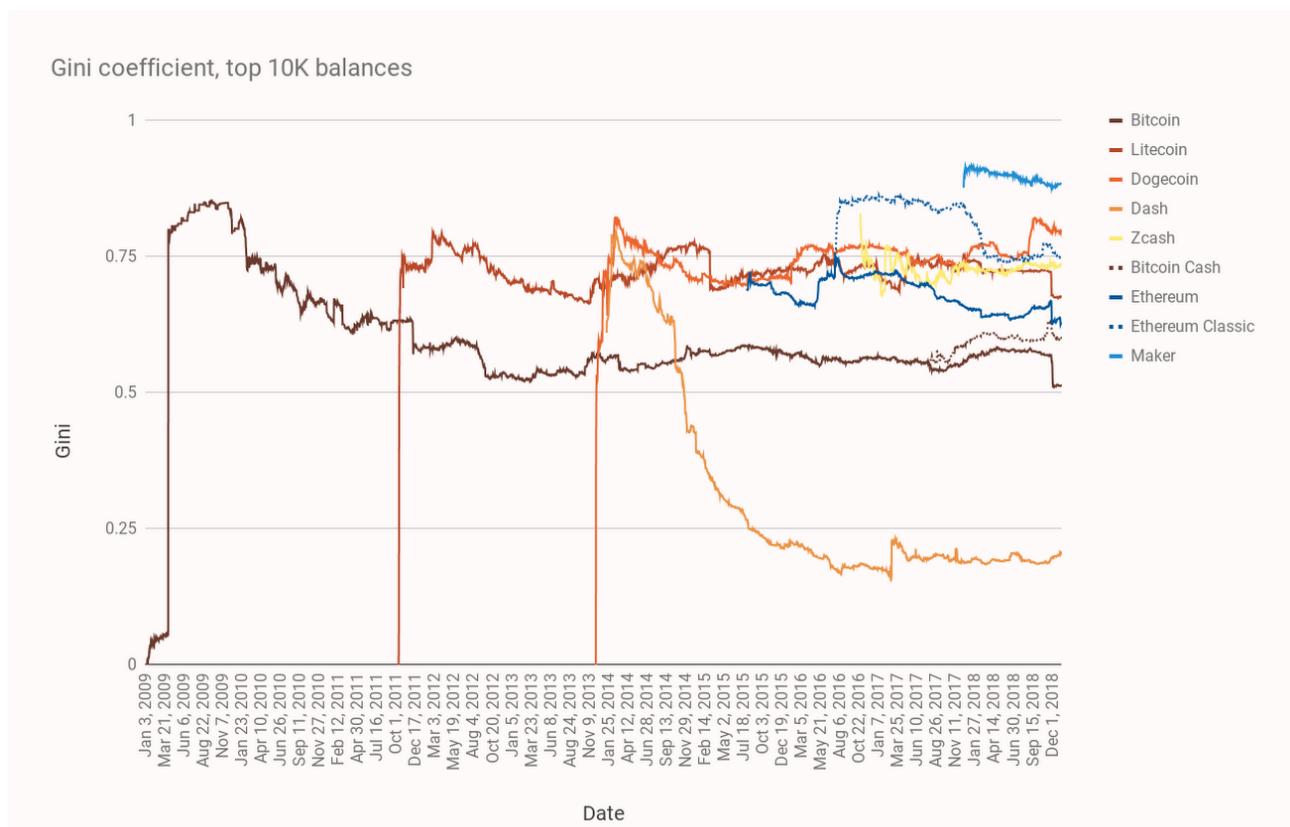


Lorenz Curve and Gini Coefficient of Edgeware Token Distribution

Caveats

- Some of the largest distributions are from [exchanges like Binance](#). These accounts signaled on behalf of their users, and have announced plans to distribute the allocations to the appropriate ETH holders - so their holdings don't actually correspond to the individuals. This creates a bias towards a higher gini coefficient than the network actually possesses.
- Multiple addresses may be mapped to the same holder - which could affect the final coefficient in either direction - it is impossible to tell which accounts may or may not be in the control of the same person.
- Most importantly, Gini coefficient analysis is sensitive to small amounts, of which Edgeware's distribution has a high number.

- For our final analysis of the Edgeware distribution, we modeled the [Google BigQuery network analysis](#) conditions, in order to create a more comparable set of results to existing networks. As Edgeware, being new, doesn't have anywhere near 10000 addresses, we can find use a percentage as a relative measure. [Coinmetrics.io reports that at the time of Google's analysis](#), active account numbers for Ethereum, for instance, had a total of 233.701k addresses. As Google looked at the top 10,000 addresses, they examined the top 4.2% of accounts.
- For the sake of comparison then, the top 4.2 percent of Edgeware addresses form the basis of [our dataset](#). In order to find the gini coefficient, we entered the values of these accounts into a [simple gini calculator](#). This provides the coefficient for roughly 90% of all lockdrop-distributed Edgeware tokens. Let's compare Edgeware's gini of 0.694 to top cryptonetworks as found by Google BigQuery:



Gini Coefficients of Popular Cryptonetworks (Top 10k Addresses) measured by Google BigQuery

If we focus on the initial launch times of these networks, no network with less than 0.55, placing the **Edgeware lockdrop distribution firmly as the most decentralized token**

distributions. (Google notes in their analysis that Dash's numbers may be unreliable due to privacy features.)

With the lockdrop contracts proven to both function and to be secure, we expect future uses of the mechanism to be even fairer as bonus structures are improved and users trust contracts to participate in longer locktimes.

Long Term Interest

In ICOs or other methods to distribute tokens, capital begets capital. However, in a lockdrop, users can be increased through time preference—bonuses for participating longer. In this way, it allowed many small ETH amount holders to show their preference. The overwhelming proportion of lockers chose not to lock for 3 months, but the maximum amount of time, 12 months. This shows us that they have a long term interest in the network. Moreover, individuals signaled their participation in additional ways, with over 130 addresses indicating that they'd like to validate. This amount has been backed up by the more than ~70 validators participating over the course of this last testnet.

Continued Governance Experimentation

by Dillon Chen

As we discussed before launching the lockdrop. One goal for us was to further experimentation around governance. It's safe to say that a few seeds have been planted. First, we've seen many teams working on further experiments in token distribution—for example, NuCypher and the [Worklock](#). By adding a covenant and allocating only to those who are validating *and* adding slashing in things get much more interesting.

This experimentation wasn't limited to other projects, the same lockdrop allocation was also used to spin up a separate network, Straightedge. We're excited to see this develop.

On [Commonwealth](#), there have been community participants suggesting that we could make the lockdrop more than just a single asset. This is something that the Edgeware governance system could vote on post-launch for any future EDG distributions. Or a further experiment.

Governance is an ongoing process, with that we've built tooling for EDG holders to easily participate in this process community to *fully* utilize the token:

- Vote on council members to help shape the future direction of Edgeware.
- Delegate EDG to validators and bond for their own validator positions to help secure the Edgeware network.
- Vote on treasury proposals to help fuel ongoing experiments, propel adoption, and reward the network's most active contributors.
- Climate DAOs
- [Funding the ambassador program](#)
- Hosting a worldwide **EdgeCon**

What does this lead us?

There is quite a bit of uncharted territory for the community to explore. There are more than a few proposals that we're happy to share on Commonwealth. In another few months, we'll be sharing updates on the improvements the community has achieved!

See [Commonwealth.im](#) for existing proposals.

Contract & Specifications

The lockdrop contract is a simple two function smart contract. When the function `lock` is called with the interface specified below via the Master Contract Lockdrop, a new Lockdrop User Contract is created that holds the timelocked ETH. Note, when `isValidator` is yes, the total allocated EDG may be staked at the time of distribution.

Notably, the Master Lockdrop Contract (MLC) itself does not hold the totality of locked ETH, instead, when calling `lock`, an individual Lockdrop User Contract (LUC) is created which holds the participant's time-locked ETH. This 2-step arrangement reduces the potential value and feasibility of a malicious attack. Further, the Master Lockdrop and Lockdrop User Contracts are extremely simple.

In particular, the LUC contract is forty-five instructions until completing the call with no jumps. These have been audited by a third-party, Quantstamp.

Lockdrop Participation

The Master Lockdrop Contract will accept "locks" and "signals" for a ninetyday Contribution Period. On Edgeware, lock interactions may have a duration of three months, six months, or 12 months with bonuses of 0%, 30%, and 120% respectively. In order to reduce centralization of power on Edgeware and increase the diversity of stakeholder voice, no single contributing ETH address or receiving EDG address will be able to obtain greater than or equal to 20% of the total EDG minted through the Lockdrop.

Advanced Topics

Setup Multi Signature Account

It is possible to create a multi-signature account in Substrate based chains like Edgeware. A multi-signature account is composed of one or more addresses and a threshold. The threshold defines how many signatories (participating addresses) need to agree on the submission of an extrinsic in order for the call to be successful.

For example, Alice, Bob, and Charlie set up a multi-sig with a threshold of 2. This means Alice and Bob can execute any call even if Charlie disagrees with it. Likewise, Charlie and Bob can execute any call without Alice. A threshold is typically a number smaller than the total number of members but can also be equal to it, which means they all have to be in agreement.

Multi-signature accounts have several uses:

- **securing your own stash:** use additional signatories as a 2FA mechanism to secure your funds. One signer can be on one computer, another can be on another, or in cold storage. This slows down your interactions with the chain, but is orders of magnitude more secure.
- **board decisions:** legal entities such as businesses and foundations use multi-sigs to collectively govern over the entity's treasury.
- **group participation in governance:** a multi-sig account can do everything a regular account can. A multi-sig account could be a council member in Kusama's governance, where a set of community members could vote as one entity.

Multi-signature accounts cannot be modified after being created. Changing the set of members or altering the threshold is not possible and instead requires the dissolution of the current multi-sig and creation of a new one. As such, multi-sig account addresses are **deterministic**, i.e. you can always calculate the address of a multi-sig just by knowing the members and the threshold, without the account existing yet. This means one can send tokens to an address that does not exist yet, and if the entities designated as the recipients come together in a new multi-sig under a matching threshold, they will immediately have access to these tokens.

Resources

- How to Create and Use Multisigs
- Learn more about Multi Signature accounts
- Generate a Multisig Account from code

Create Multi Signature Account

In this chapter we will create a Multi-Signature account and perform a test transaction from the account.

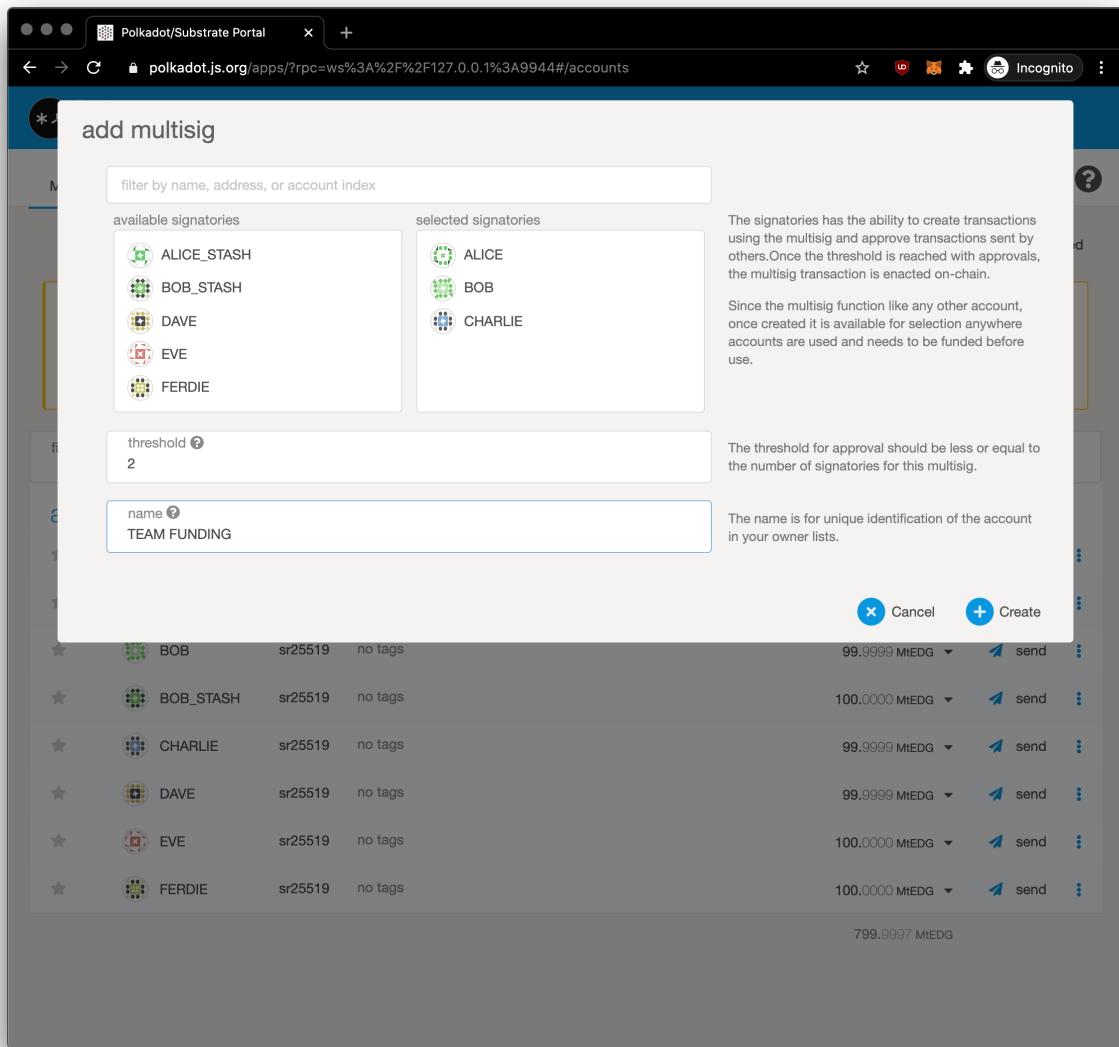
You will go to the [Accounts in Apps](#)

The screenshot shows the 'Accounts' tab selected in the top navigation bar of the Polkadot/Substrate Portal. Below the navigation, there are two buttons: 'My accounts' (selected) and 'Vanity generator'. A yellow box highlights the 'Multisig' button among other account management options like 'Add account', 'Restore JSON', 'Add via Qr', and 'Proxied'. A note at the top of the main content area recommends creating accounts securely and lists the 'polkadot-js extension' as a recommended browser extension. The main table displays eight accounts: ALICE, ALICE_STASH, BOB, BOB_STASH, CHARLIE, DAVE, EVE, and FERDIE, each with their type (sr25519), tags (none), and current balance in MtEDG. The total balance shown at the bottom is 799.9997 MtEDG. The table has columns for accounts, type, tags, balances, and actions (send and more).

accounts	type	tags	balances	
★ ALICE	sr25519	no tags	99.9997 MtEDG	↗ send ⋮
★ ALICE_STASH	sr25519	no tags	100.0000 MtEDG	↗ send ⋮
★ BOB	sr25519	no tags	99.9999 MtEDG	↗ send ⋮
★ BOB_STASH	sr25519	no tags	100.0000 MtEDG	↗ send ⋮
★ CHARLIE	sr25519	no tags	99.9999 MtEDG	↗ send ⋮
★ DAVE	sr25519	no tags	99.9999 MtEDG	↗ send ⋮
★ EVE	sr25519	no tags	100.0000 MtEDG	↗ send ⋮
★ FERDIE	sr25519	no tags	100.0000 MtEDG	↗ send ⋮

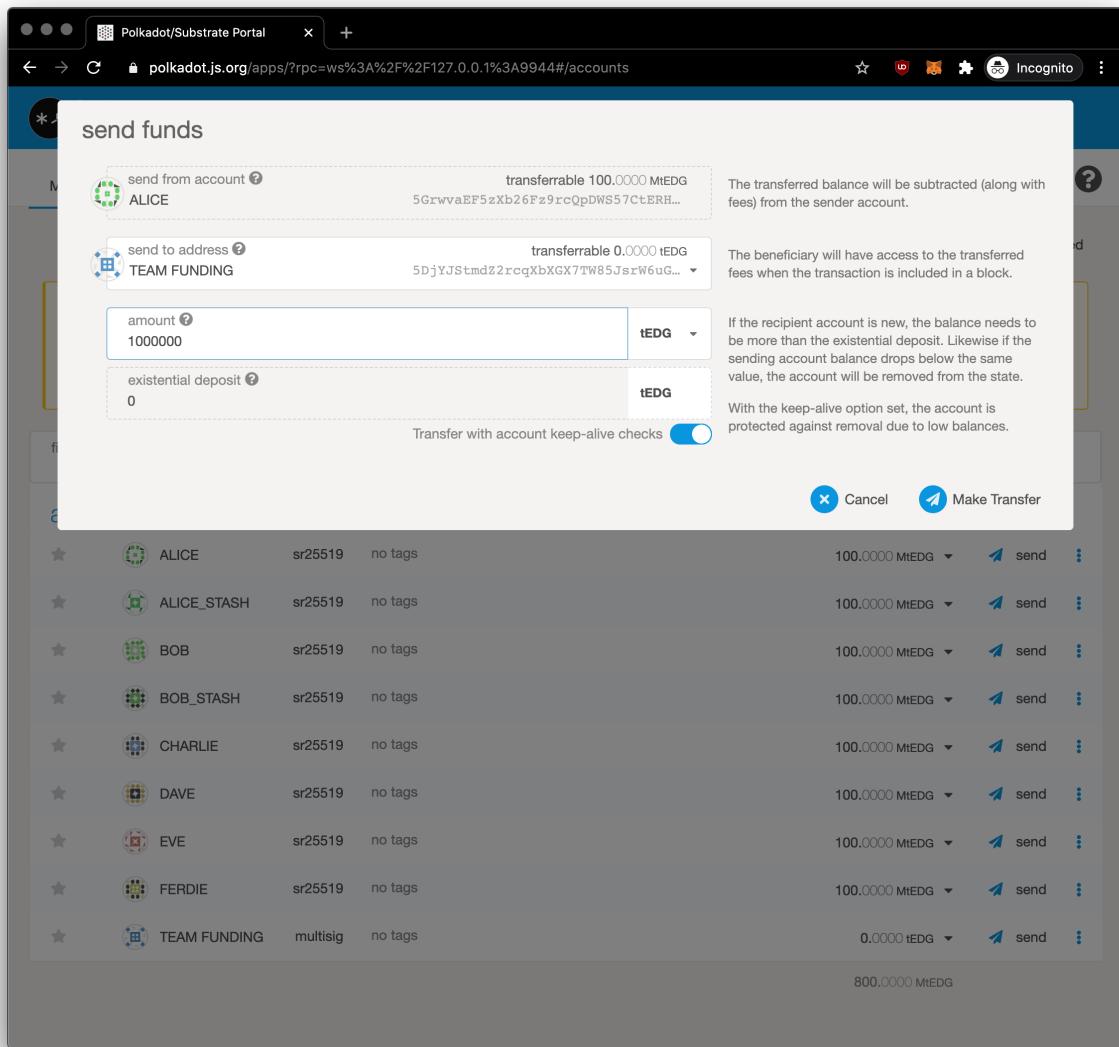
create-ms-account

You will choose your signatures, i.e. your team mates accounts, in our scenario we choose **Alice, Bob and Charlie**. We set **threshold to 2**, that means there is only needed signature from **two of three**. You can set threshold to be less or equal to the number of signatories for multisig. We call it our **Team Funding** account.



create-choose-signatures

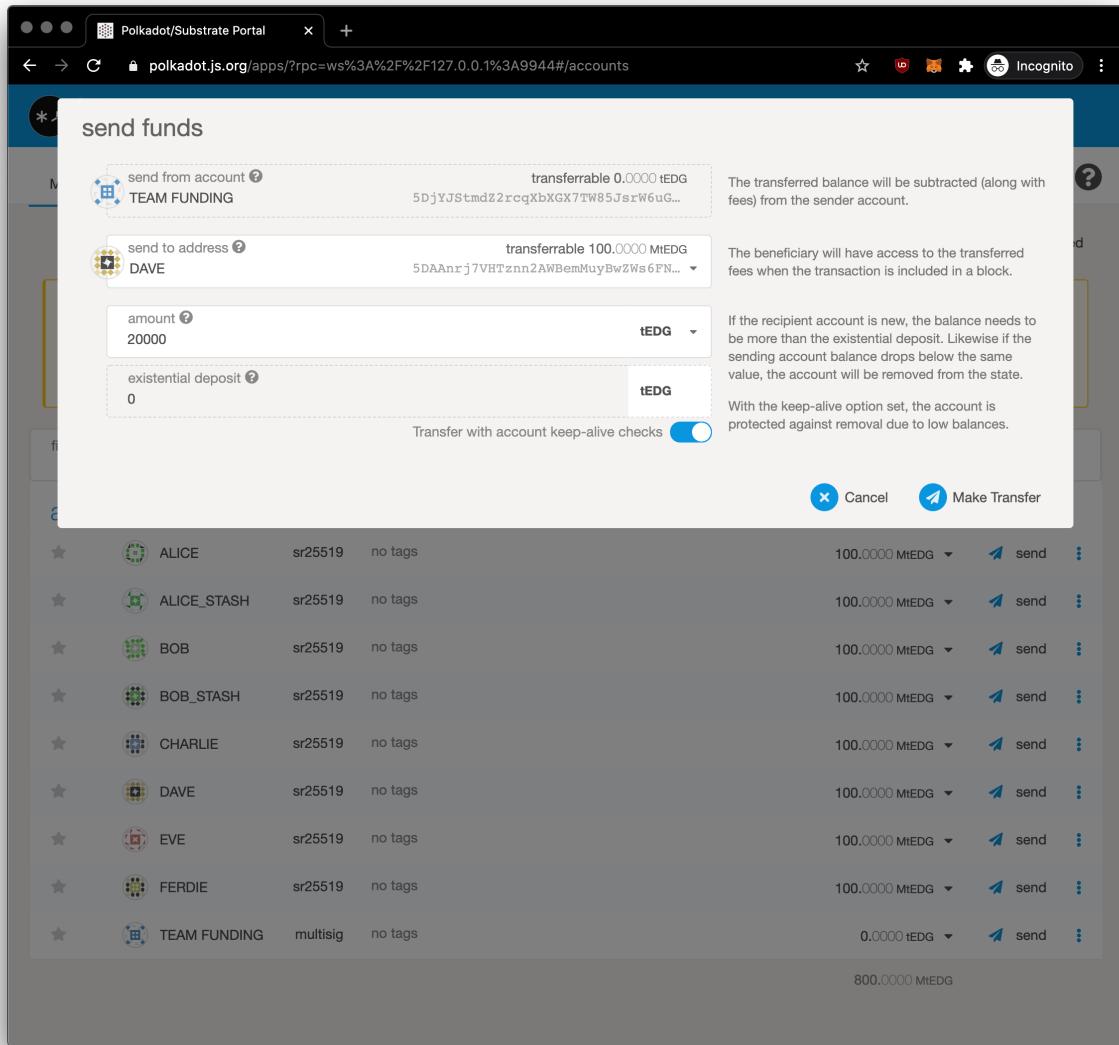
Now we will top-up **1,000,000 tEDG** to the **Team Funding** multi-signature account from **Alice**. Thank you Alice!



top-up-multisignature

Create Transfer from Multi-Signature Account

Now that the **Team Funding** account has been seeded, we can pay **DAVE 200,000 tEDG** for completion of a project. In our accounts list, hit **Send** from our **Team Funding** account



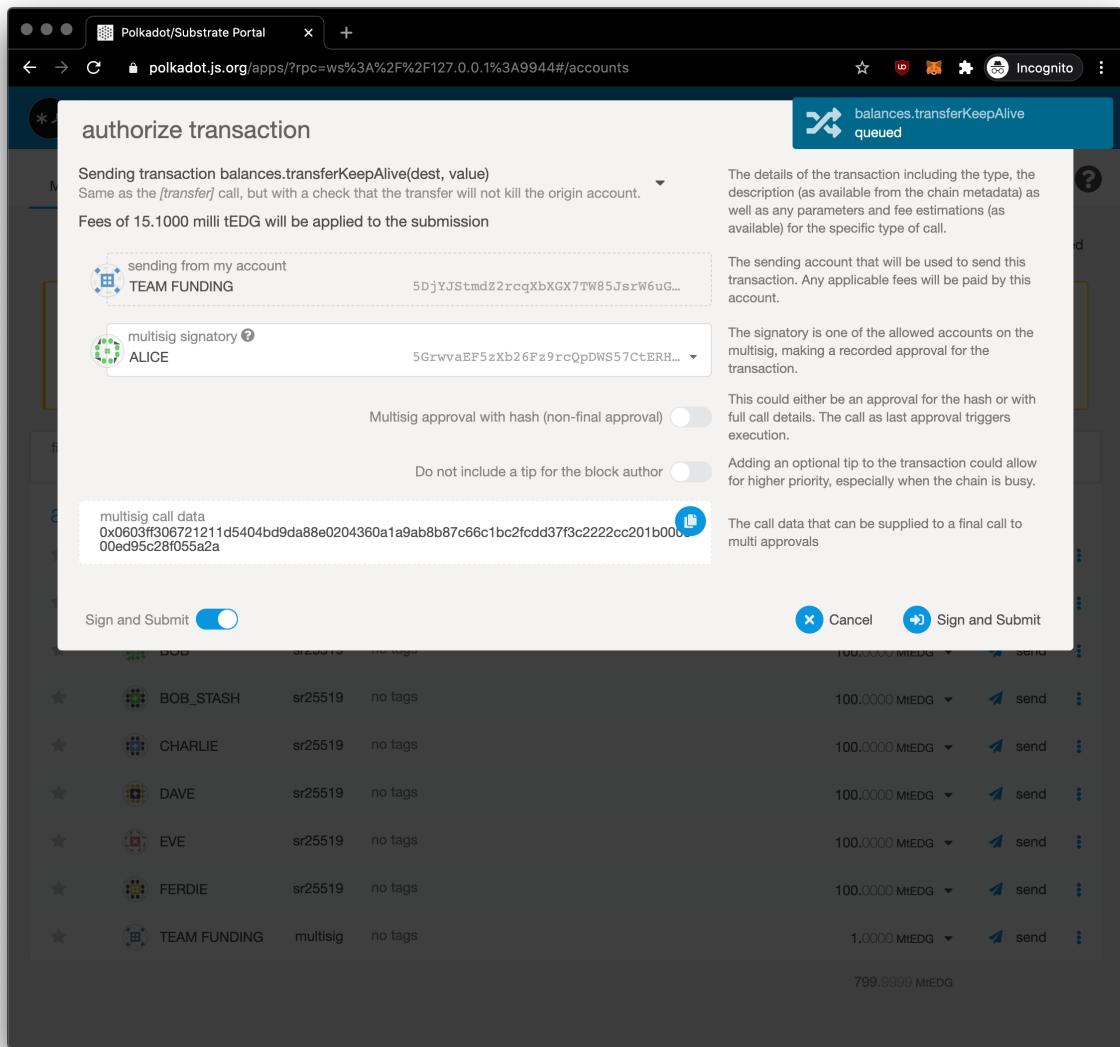
transfer-ms

Now we are prompted (as ALICE) to authorize transaction.

You will see there `multisig call data` with payload

```
0x0603ff306721211d5404bd9da88e0204360a1a9ab8b87c66c1bc2fcdd37f3c2222cc201b000000  
ed95c28f055a2a
```

which is call data that can be supplied to a final call to multi approvals. It's what triggers chain logic to execute commands. You will hit Sign and Submit.



transfer-ms-authorize-tx.png

It was broadcasted to the chain, you can see notification on the top right.

The screenshot shows the Polkadot/Substrate Portal's Accounts page. At the top, there are tabs for 'Development' (version 41, #2,700), 'Accounts' (selected), 'Network', 'Governance', and 'Developer'. A notification bar at the top right says 'Dismiss all notifications' and lists extrinsics: 'balances.transferKeepAlive inblock', 'system.ExtrinsicSuccess', 'balances.Reserved', 'multisig.NewMultisig', 'treasury.Deposit', and 'extrinsic event'. Below the tabs, there are buttons for 'Add account' and 'Restore JSON'. A section titled 'My accounts' and 'Vanity generator' is shown. A yellow box contains a note about secure account creation and a list of browser extensions: 'polkadot-js extension (Basic account injection and signer)'. Another note says 'Accounts injected from any of these extensions will appear in this application and be available for use. The above list is updated as more extensions with external signing capability become available. [Learn more...](#)'.

	accounts	type	tags	balances	
★	ALICE	sr25519	no tags	99.9996 MtEDG	send ⋮
★	ALICE_STASH	sr25519	no tags	100.0000 MtEDG	send ⋮
★	BOB	sr25519	no tags	99.9999 MtEDG	send ⋮
★	BOB_STASH	sr25519	no tags	100.0000 MtEDG	send ⋮
★	CHARLIE	sr25519	no tags	99.9999 MtEDG	send ⋮
★	DAVE	sr25519	no tags	99.9999 MtEDG	send ⋮
★	EVE	sr25519	no tags	100.0000 MtEDG	send ⋮
★	FERDIE	sr25519	no tags	100.0000 MtEDG	send ⋮
★	TEAM FUNDING	multisig	no tags	100.0000 tEDG	send ⋮
799.9997 MtEDG					

multi-signature-first-call

Now we click on the right three dots next to our **Team Funding** account as there is multisig approvals pending (*red dot*) and select **Send**

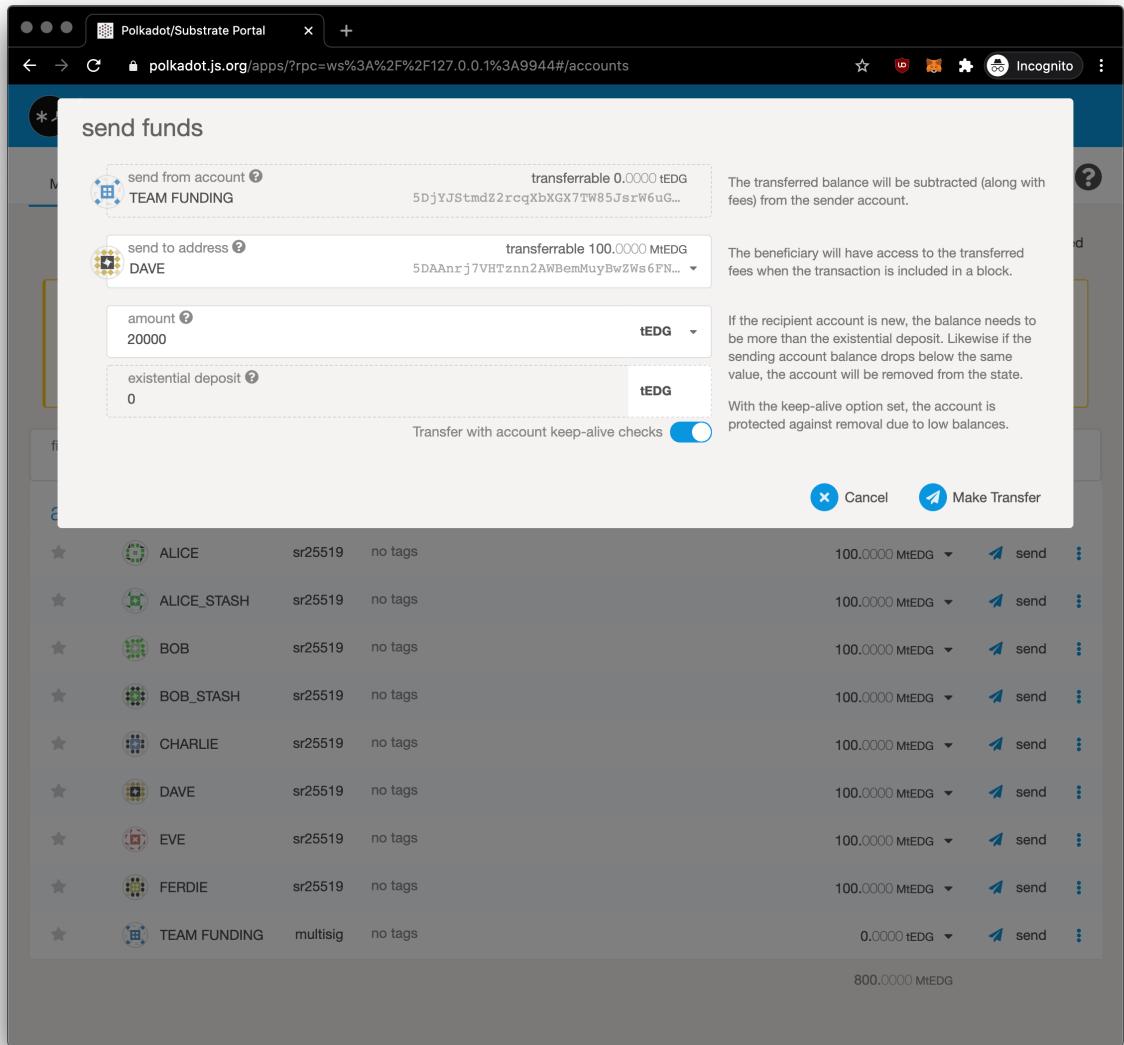
The screenshot shows the Polkadot/Substrate Portal's Accounts page. At the top, there are tabs for 'My accounts' and 'Vanity generator'. Below the tabs are several action buttons: '+ Add account', 'Restore JSON', 'Add via QR', '+ Multisig', and '+ Proxied'. A note at the top of the main content area recommends creating accounts securely and lists the 'polkadot-js extension' as a browser extension available for use. It also states that accounts injected from these extensions will be available for use. The main table displays the following account information:

	accounts	type	tags	balances	
★	ALICE	sr25519	no tags	98.9999 MtEDG	send ⋮
★	ALICE_STASH	sr25519	no tags	100.0000 MtEDG	send ⋮
★	BOB	sr25519	no tags	100.0000 MtEDG	send ⋮
★	BOB_STASH	sr25519	no tags	100.0000 MtEDG	send ⋮
★	CHARLIE	sr25519	no tags	100.0000 MtEDG	send ⋮
★	DAVE	sr25519	no tags	100.0000 MtEDG	send ⋮
★	EVE	sr25519	no tags	100.0000 MtEDG	send ⋮
★	FERDIE	sr25519	no tags	100.0000 MtEDG	send ⋮
★	TEAM FUNDING	multisig	no tags	1.0000 MtEDG	send ⋮

Total balance: 799.9999 MtEDG

multi-signature-second-call

You as other signator (**BOB**) enter same amount and destination to **get same final call payload.**



multi-signature-transfer

Now we as **BOB** we can authorize transaction. UI is smart enough and detected it's final approval. Under toggle *Multisig message* you will see same payload as we're creating multisig transaction. Hit Sign and Submit and it's should be signed 2 of 3 signatories which is enough for this scenario so can transaction can pass through to DAVE.

Polkadot/Substrate Portal

polkadot.js.org/apps/?rpc=ws%3A%2F%2F127.0.0.1%3A9944#/accounts

Incognito

authorize transaction

Sending transaction balances.transferKeepAlive(dest, value)
Same as the [transfer] call, but with a check that the transfer will not kill the origin account.

Fees of 15,1000 milli tEDG will be applied to the submission

The details of the transaction including the type, the description (as available from the chain metadata) as well as any parameters and fee estimations (as available) for the specific type of call.

sending from my account
TEAM FUNDING

multisig signatory ?
BOB

Multisig message with call (for final approval)

Do not include a tip for the block author

The sending account that will be used to send this transaction. Any applicable fees will be paid by this account.

The signatory is one of the allowed accounts on the multisig, making a recorded approval for the transaction.

This could either be an approval for the hash or with full call details. The call as last approval triggers execution.

Adding an optional tip to the transaction could allow for higher priority, especially when the chain is busy.

Sign and Submit

CANCEL SIGN AND SUBMIT

Account	Address	Tags	Balance	Action	More
ALICE_STASH	sr25519	no tags	100,0000 MIEDG		
BOB	sr25519	no tags	100,0000 MtEDG		
BOB_STASH	sr25519	no tags	100,0000 MtEDG		
CHARLIE	sr25519	no tags	100,0000 MtEDG		
DAVE	sr25519	no tags	100,0000 MtEDG		
EVE	sr25519	no tags	100,0000 MIEDG		
FERDIE	sr25519	no tags	100,0000 MtEDG		
TEAM FUNDING	multisig	no tags	1,0000 MtEDG		
			799,9999 MIEDG		

multi-signature-call-confirm

Woala, DAVE has money, **funds secured** on his account.

The screenshot shows the Polkadot/Substrate Portal's Accounts page. At the top, there are tabs for 'Development version 41 #69', 'Accounts', 'Network', 'Governance', and 'Developer'. A tooltip for the 'balances.transferKeepAlive inblock' extrinsic is open, listing other extrinsics: 'system.ExtrinsicSuccess', 'balances.Unreserved', 'balances.Transfer', 'multisig.MultisigExecuted', 'treasury.Deposit', and 'extrinsic event'. Below the tooltip, a note says it's recommended to create/store accounts securely and externally from the app, mentioning the polkadot-js extension.

accounts	type	tags	balances
ALICE	sr25519	no tags	98,9999 MtEDG
ALICE_STASH	sr25519	no tags	100,0000 MtEDG
BOB	sr25519	no tags	99,9999 MtEDG
BOB_STASH	sr25519	no tags	100,0000 MtEDG
CHARLIE	sr25519	no tags	100,0000 MtEDG
DAVE	sr25519	no tags	100,2000 MtEDG transferrable 100,2000 MtEDG
EVE	sr25519	no tags	100,0000 MtEDG
FERDIE	sr25519	no tags	100,0000 MtEDG
TEAM FUNDING	multisig	no tags	800,000,0000 tEDG

799,9999 MtEDG

multi-signature-funds-secured

You've managed to learn how to make transaction for multi-signature. Multisignature has broad usecase and you can leverage final call for your use case to trigger what ever you want on chain.

Making Transaction with a multi-signature account

There are three types of actions you can take with a multi-sig account:

- Executing a call.
- Approving a call.
- Cancelling a call.

In scenarios where only a single approval is needed, a convenience method `as_multi_threshold_1` should be used. This function takes only the other signatories and the raw call as its arguments.

However, in any case besides a single approval, you will likely need more than one of the signatories to approve the call before finally executing it. When you create a new call or approve a call as a multi-sig, you will need to place a small deposit. The deposit stays locked in the pallet until the call is executed. The reason for the deposit is to place an economic cost on the storage space that the multi-sig call takes up on the chain and discourage users from creating dangling multi-sig operations that never get executed. The deposit will be reserved in the caller's accounts so participants in multi-signature wallets should have spare funds available.

The deposit is dependent on the threshold parameter and is calculated as follows:

```
Deposit = DepositBase + threshold * DepositFactor
```

Where `DepositBase` and `DepositFactor` are chain constants set in the runtime code.

Currently, the `DepositBase` is equal to `deposit(1, 88)` and the `DepositFactor` is equal to `deposit(0,32)`.

Utilities

Generating Addresses of Multi-signature Accounts

NOTE: Addresses that are provided to the multi-sig wallets must be sorted. The below methods for generating sort the accounts for you, but if you are implementing your own sorting then be aware that the public keys are compared byte-for-byte and sorted ascending before being inserted in the payload that is hashed.

Addresses are deterministically generated from the signers and threshold of the multisig wallet. The [w3f/msig-util](#) is a small CLI tool that can determine the multisignature address based on your inputs.

You can see here, we took addresses of development accounts ALICE, BOB and CHARLIE and we've got same address, that we have in our Apps.

```
1 ➤ npx @w3f/msig-util derive --addresses 5GrwvaEF5zXb26Fz9rcQpDWS57CtERHpNehXCPcNoHGKutQY 5FHneW46xGXgs5mLjZPvHJGJyfR
2 -----
3 Addresses: 5GrwvaEF5zXb26Fz9rcQpDWS57CtERHpNehXCPcNoHGKutQY 5FHneW46xGXgs5mLjZPvHJGJyfR
4 Threshold: 2
5 Multisig Address (SS58: 42): 5DjYJStmdZ2rcqXbXGX7TW85JsrW6uG4y9MUcLq2BoPMpR
6 -----
```

- [w3f/msig-util - internal code](#)

Setting Up a Public UI

This page explains how to set up a [polkadot-js/apps UI](#) for Edgeware, using the Edgeware branch of `apps`. The main difference is that the Edgeware branch supports the types defined in Edgeware's voting and governance modules.

0. Provisioning a server

Provision an appropriately sized server from one of the recommended VPS providers.

We assume you are using Ubuntu 18.04 x64; other versions or operating systems will require adjustments to these instructions.

Set up DNS pointing to the server, from e.g. `apps.edgewa.re`. It is strongly recommended that you do this now.

SSH into the server.

1. Installing `apps` and setting it up as a system service

Clone the `apps` repo:

```
1 git clone https://github.com/hicommonwealth/apps.git
2 cd apps
```

Install nodejs 11.x and yarn. You will need to add new package repositories:

```
1 curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -
2 echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee /etc/apt/se
3 curl -sL https://deb.nodesource.com/setup_11.x | sudo -E bash -
4 apt update
```

```
5 apt install -y nodejs yarn
```

Install dependencies:

```
yarn
```

Create the apps service:

```
1 {
2     echo '[Unit]'
3     echo 'Description=EdgewareApps'
4     echo '[Service]'
5     echo 'Type=exec'
6     echo 'WorkingDirectory='`pwd`'
7     echo 'Environment=ENV=production'
8     echo 'ExecStart=yarn run start'
9     echo '[Install]'
10    echo 'WantedBy=multi-user.target'
11 } > /etc/systemd/system/apps.service
```

Start and check the service is running:

```
1 systemctl start apps
2 systemctl status apps
3 curl localhost:3000
```

2. Configuring an SSL certificate

To make `apps` available in a secure manner, we will use Let's Encrypt and certbot to make the server available via SSL.

Install Certbot:

```
1 apt -y install software-properties-common  
2 add-apt-repository universe  
3 add-apt-repository ppa:certbot/certbot  
4 apt update  
5 apt -y install certbot python-certbot-nginx
```

Run certbot to get a certificate from Let's Encrypt:

```
certbot --nginx
```

Certbot will ask you some questions, start its own web server, and talk to Let's Encrypt to issue a certificate.

When it asks you whether to redirect traffic from port 80 to SSL, you should respond **yes**.

3. Updating the nginx configuration

Set the intended public address of the server, e.g. apps.edgewa.re, as an environment variable:

```
export name=apps.edgewa.re
```

Set up an nginx configuration. This will inject the public address you have just defined.

```
1 {  
2     echo 'user      www-data; ## Default: nobody'  
3     echo 'worker_processes 5; ## Default: 1'  
4     echo 'error_log  /var/log/nginx/error.log;'  
5     echo 'pid      /var/run/nginx.pid;'  
6     echo 'worker_rlimit_nofile 8192;'  
7     echo ''  
8     echo 'events {'
```

```
9     echo '    worker_connections  4096; ## Default: 1024'
10    echo '}'
11    echo ''
12    echo 'http {'
13    echo '      map $http_upgrade $connection_upgrade {'
14    echo '          default upgrade;'
15    echo "          \\" close;""
16    echo '      }'
17    echo '      server {'
18    echo '          listen      443 ssl;'
19    echo '          server_name '$name';'
20    echo ''
21    echo '          ssl_certificate /etc/letsencrypt/live/'$name'/cert.pem;'
22    echo '          ssl_certificate_key /etc/letsencrypt/live/'$name'/privkey.pem;'
23    echo '          ssl_session_timeout 5m;'
24    echo '          ssl_protocols  SSLv2 SSLv3 TLSv1;'
25    echo '          ssl_ciphers  HIGH:!aNULL:!MD5;'
26    echo '          ssl_prefer_server_ciphers  on;'
27    echo ''
28    echo '          location / {'
29    echo '              proxy_pass http://127.0.0.1:3000 ;'
30    echo '          }'
31    echo '      }'
32    echo '}'
33 } > /etc/nginx/nginx.conf
```

Start nginx, and ensure it is run on system startup:

```
1 systemctl daemon-reload
2 systemctl start nginx
3 systemctl enable nginx
```

Check your server is running from your local browser.

Validating on Edgeware

Welcome to the official, in-depth Edgeware guide to validating. We're happy that you're interested in validating on Edgeware and we'll do our best to provide in-depth documentation on the process below. As always, reach out on [Discord](#) or [Telegram](#) if you have questions about the project.

This document contains all the information one should need to start validating on Edgeware using the [polkadot-js/apps user interface](#). We will start with how to setup one's node and proceed to how to key management and monitoring. To start, we will use the following terminology of keys for the guide:

- **stash** - the stash keypair is where most of your funds should be located. It can be kept in cold storage if necessary.
 - **controller** - the controller is the keypair that will control your validator settings. It should have a smaller balance, e.g. 10-100 EDG
 - **session** - the 3 session keypairs are hot keys that are stored on your validator node. They do not need to have balances.
-

Requirements

1. You should have balances in your `stash` (ed25519 or sr25519) and `controller` (ed25519 or sr25519) accounts.
2. Instructions for setting up a node are [here](#). You will need to additionally add the `--validator` flag to run a validator node.
3. You should have a wallet, such as the `polkadot-js` extension, installed in your browser with the stash and controller keypairs. If you don't have it, get it [here](#).

If you need to request a testnet EDG balance, just ask on [Discord](#).

1. Install the Edgeware node

If you are starting a validator node that needs to stay up, we recommend [following the default instructions](#) and [setting up monitoring](#).

Then, you should go to the line where `target/release/edgeware` is called in `/etc/systemd/system/edgeware.service`, and add the `--validator` flag. Reload the service configuration and check to see the node has started properly:

```
1 systemctl daemon-reload  
2 systemctl restart edgeware  
3 systemctl status edgeware
```

You should see this output:

```
1 2019-10-03 10:28:59 Edgeware  
2 2019-10-03 10:28:59 version 1.0.0-3f34fba-x86_64-macos  
3 2019-10-03 10:28:59 by Commonwealth Labs, 2018-2019  
4 2019-10-03 10:28:59 Chain specification: Edgeware  
5 2019-10-03 10:28:59 Node name: naughty-light-7646  
6 2019-10-03 10:28:59 Roles: AUTHORITY
```

Make sure the Chain specification is correct, and the node is running with

Roles: AUTHORITY. If you don't see the correct Chain, you should provide the correct `--chain` parameter, and if you don't see the Authority role, you should make sure you're actually running Edgeware with the `--validator` flag.

Make sure you can access the node through the command line. If you enter the `curl` command below, you should see this output:

```
1 curl --include --no-buffer --header "Connection: Upgrade" --header "Upgrade  
2  
3 HTTP/1.1 101 Switching Protocols  
4 Connection: Upgrade  
5 Sec-WebSocket-Accept: qGEgH3En71di5rrssAZTmtRTyFk=  
6 Upgrade: websocket
```

Check the [telemetry dashboard](#), and make sure your node is syncing up to the latest block. If any of these steps do not check out, stop now; you will not be able to validate until they are

corrected.

2. Connect via the interface

Go to the [polkadot-js web interface](#) and connect to a custom node, e.g. testnet1.edgewa.re, testnet2.edgewa.re, or testnet3.edgewa.re.

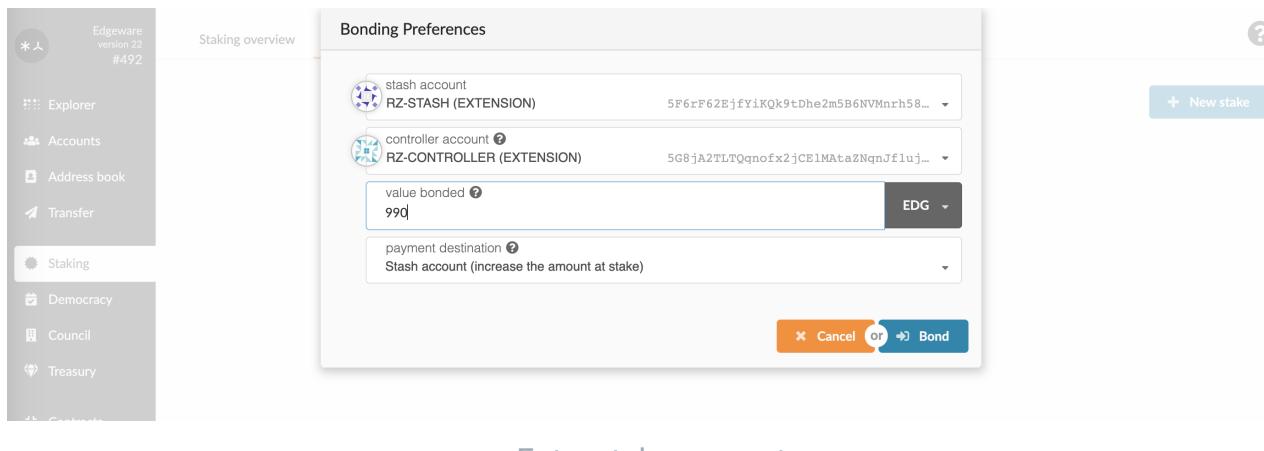
The interface should show the correct latest block.

3. Create a stake

Go to the **Staking** tab, and select **Account actions** at the top. Click on **New stake**.

Select your controller and stash accounts. Enter how much of your stash balance you would like to stake. Leave a few EDG free, or you will be unable to send transactions from the account.

You can also choose where your validator rewards are deposited (to the stash or the controller) and whether rewarded EDG should be automatically re-staked.



Sign and send the transaction.

4. Set your session keys, using `rotateKeys`

Click on **Set Session Keys** on the stake you just created above.

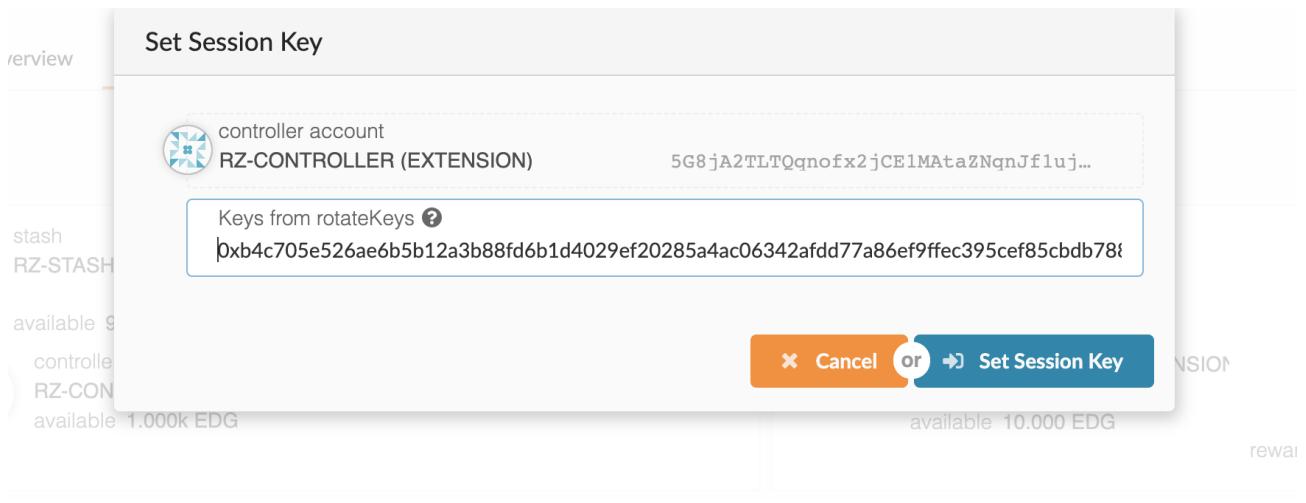
Go to the command line where your validator is running (e.g. SSH into the server, etc.) and enter this command. It will tell your validator to generate a new set of session keys:

```
curl -H 'Content-Type: application/json' --data '{ "jsonrpc":"2.0", "method":
```

The output should look like this:

```
{"jsonrpc":"2.0", "result":"0x0ca0fbf245e4abca3328f8bba4a286d6cb1796516fcc6886
```

Copy the hexadecimal key from inside the JSON object, and paste it into the web interface.



rotateKeys input

Sign and send the transaction.

5. Start validating

stash
RZ-STASH (EXTENSION)
available 9.974 EDG

controller
RZ-CONTROLLER (EXTENSION)
available 999.969 EDG

bonded 990.000 EDG
session keys 0xb4c705...14a702
reward destination staked

Validate or nominate

You should now see a **Validate** button on the stake. Click on it, and enter the commission you would like to charge as a validator. Sign and send the transaction.

You should now be able to see your validator in the **Next up** section of the staking tab.

At the beginning of the next **era**, if there are open slots and your validator has adequate stake supporting it, your validator will join the set of active validators and automatically start producing blocks. (On the testnet, sessions are 100 blocks or 10 minutes long, and eras are 300 blocks or 30 minutes long.)

Active validators receive rewards at the end of each era. Slashing also happens at the end of each era.:

Chain info Block details Node info

block hash or number to query

600

parentHash: 0x7ba57bcd3d1a852c697dafc6b86741b6a2359f6798c983c61e5aca7de294c54
extrinsicsRoot: 0x4fba4e6d3edbbe4d0c13a7c83b976987bd675dc57a764a2579df326267b98d87
stateRoot: 0x91b9602b15a8e4d3c8e198ad33449f4d7f5c05902d9ad75ccca96870fe91d7f3

extrinsics

timestamp.set (#0)
Set the current time. This call should be invoked exactly once per block. It will pa...

events

staking.Reward (#0)
All validators have been rewarded by the given balance.
Balance: 1.931 Kilo

session.NewSession (#0)
New session has happened. Note that the argument is the session index, not the ...

system.ExtrinsicSuccess (#0)
An extrinsic completed successfully.

treasuryReward.TreasuryMinting

logs

Reward

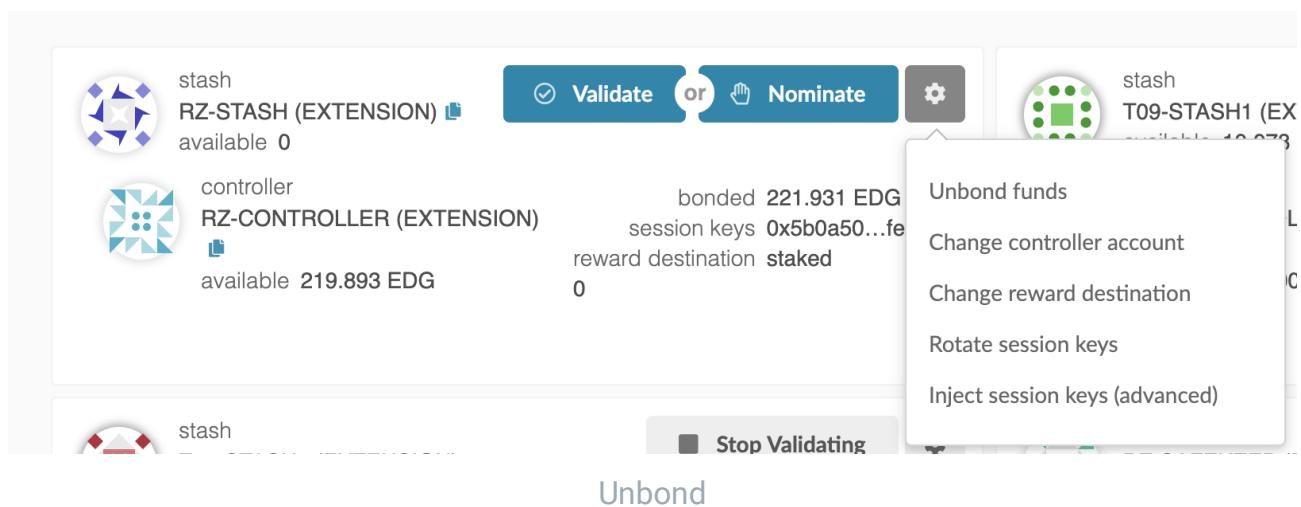
Is your validator not producing blocks?

- Check that it is part of the active validator set. You will need to wait until your validator rotates in; this may take longer depending on whether there are free slots.
- Check that it is running with the `--validator` flag.
- Ensure your session keys are set correctly. Use `curl` to rotate your session keys again, and then send another transaction to the network to set the new keys.

6. Stop validating

If you would like to stop validating, you should use the **Stop Validating** button on your stake, to send a `chill` transaction. It will take effect when the next validator rotation happens, at which point you can shut down your validator.

Once you have stopped validating, you can send a transaction to unbond your funds. You can then **redeem** your unbonded funds after the unbonding period has passed.





stash

RZ-STASH (EXTENSION) 

available 0

 Validate

or

 Nominate



controller

RZ-CONTROLLER (EXTENSION) 

available 219.871 EDG

Redeem these funds

bonded 21.931 EDG

redeemable 200.000 EDG 

session keys 0xb0a50...fe6210

reward destination staked

0

Redeem unbonded funds

Setting Up Monitoring

If you are running a validator or a public node, you should set up system monitoring to be aware if your node goes offline, and restart it automatically if possible.

This tutorial explains how to set up **monit** and **mmonit** to accomplish this:

- Monit is a process monitoring tool, which can restart your node if it stalls.
- Mmonit is a dashboard that shows the performance (CPU, memory, alerts, etc.) of monit nodes.

(If you don't need a dashboard, you can skip directly to the section for setting up `monit` on an individual node.)

The tutorial assumes you are running Ubuntu 18.04.

Setting up an **mmonit** monitoring server

For the monitoring server, a small server should be enough (e.g. 1GB memory). The default setup will use SQLite as a database. This means if you remove the monit directory, all logged events will be lost!

```
1 apt install -y monit
2 wget https://mmonit.com/dist/mmonit-3.7.3-linux-x64.tar.gz
3 tar -vxzf mmonit-3.7.3-linux-x64.tar.gz
4 mv mmonit-3.7.3 mmonit
```

Use monit to keep mmonit up:

```
1 {
2     echo 'check process mmonit with pidfile ``pwd``/mmonit/logs/mmonit.pid'
3     echo '    start program = ``pwd``/mmonit/bin/mmonit"'
4     echo '    stop program = ``pwd``/mmonit/bin/mmonit stop"'
5     echo 'set httpd port 2812 and use address localhost'
```

```
6     echo ' allow localhost'
7 } > /etc/monit/monitrc
8
9 monit reload
10 monit validate
```

To start the mmonit dashboard manually, you can use the command `mmonit/bin/mmonit`. To stop, use `mmonit/bin/mmonit stop`.

Now, go to the node in your browser, e.g. `http://monitor.edgeware.re:8080/`. Log in with username `admin` and password `swordfish` and change the default username and password.

You should end up with two users; admin is what you'll use to log in, and the other user is what you'll provide to nodes that push data to the monitoring server.

Setting up `monit` on an individual node

SSH into your node.

Before proceeding, set an appropriate hostname for the node, e.g.

```
hostnamectl set-hostname validator1
```

Install monit.

```
apt install -y monit
```

Set up a Slack webhook (optional):

```

1 export WEBHOOK=[slack webhook url]
2 {
3     echo 'URL="'$WEBHOOK"'
4     echo "PAYLOAD='{"text": \"`hostname` restarted edgeware.service\"}'"
5     echo 'curl -s -X POST --data-urlencode "payload=$PAYLOAD" $URL'
6 } > /root/slack_notify.sh
7
8 chmod 755 /root/slack_notify.sh

```

Set up a Monit config to check the Edgeware service at 10-second intervals, restart if CPU > 90% for five checks (~50sec), and post events to mmonit.

If you are using mmonit, include the username and password you have set for mmonit above. Otherwise, you should remove the sections that use the username and password below:

```

1 export USER=edgeware
2 export PASSWORD=[password]
3 export TARGET=[domain]
4 {
5     echo 'set daemon 10'
6     echo 'set log /var/log/monit.log'
7     echo 'set idfile /var/lib/monit/id'
8     echo 'set statefile /var/lib/monit/state'
9     echo 'set eventqueue'
10    echo '  basedir /var/lib/monit/events'
11    echo '  slots 100'
12    echo ''
13    echo 'check process edgeware matching target/release/edgeware'
14    echo '  start program = "/bin/systemctl restart edgeware"'
15    echo '  stop program = "/bin/systemctl kill edgeware"'
16    echo '  if cpu > 90% for 20 cycles then exec "/bin/systemctl stop edgeware"'
17    echo '  if cpu > 90% for 64 cycles then exec "/bin/systemctl kill edgeware"'
18    echo '  if cpu > 90% for 64 cycles then alert'
19    echo '  if does not exist for 1 cycles then start'
20    echo '  if does not exist for 1 cycles then exec "/bin/bash -c /root/slack_notify.sh"'
21    echo ''
22    echo 'set mmonit http://'$USER:$PASSWORD@$TARGET':8080/collector'
23    echo 'set httpd port 2812 and use address localhost'
24    echo '  allow localhost'
25    echo '  allow '$USER:$PASSWORD
26 } > /etc/monit/monitrc
27
28 chmod 600 /etc/monit/monitrc

```

```
29  
30 monit reload  
31 monit validate
```

Refer to the [monit documentation](#) if you would like to set up email alerts or other additional checks on your node.

Monitoring API health

So far, monit will only check to ensure that the node does not remain at 100% CPU for extended periods of time. We can now add another check that uses the API to ensure the node is accepting connections and syncing properly.

This will catch some failure scenarios where the node appears to keep operating but stops recognizing new blocks. We provide `nodeup` for this purpose.

Clone `nodeup` into the `/root` directory. Follow its installation instructions:

```
1 git clone https://github.com/hicommonwealth/nodeup.git  
2 cd nodeup  
3 apt install -y nodejs npm  
4 npm install -g yarn  
5 yarn
```

Make sure your node is running with the `--rpc-cors="*"` flag, so WebSocket connections are accepted. Note that by default, this will not make the node accept connections from outside the local machine (`--ws-external` is required for that).

Test nodeup:

```
node index.js
```

If it is working, you can now add these lines to your monit configuration at

/etc/monit/monitrc (they assume that nodeup is installed in the /root/nodeup directory, adjust accordingly if not):

```
1 check program nodeup with path "/root/nodeup/index.js -u ws://localhost:994"
2     if status > 0 for 10 cycles then exec "/bin/systemctl stop edgeware" and
```

Restart monit, and check that the new script is working:

```
1 monit reload
2 monit status
```

Running Cross Chain Protocols

Cross Chain Protocols

- How to port a token over (directly mirroring state)
- Having wrapped tokens, what to tell an exchange?

Ethereum <>> Edgeware Bridge

In addition to connecting to other parachains, Edgeware maintains several direct bridges, first targeting EVM-compatible chains.

XCMP

Edgeware is a project within the Polkadot Ecosystem. This allows you to run cross-chain applications, and benefit from shared security.

What is Polkadot

- What is Polkadot

What is a Parachain

A Parachain is a project within the Polkadot Ecosystem that leverages the shared security of a relay chain.

Other parachains include.

Applications may include.

Parachain to Parachain Interaction

Interacting with SPREE

Zero Knowledge Primitives

Primitives

Edgeware includes a number of zero-knowledge primitives at the runtime level. This lets developers explore new use cases for private transactions and more without having to worry about onerous fees being passed onto the user.s

Other Resources

FAQs

FAQs

How do I retrieve my ETH after the Lockdrop is over?

Use the [Unlock tool at Commonwealth.im](#)

Read the full instructions at [blog.edgewa.re](#)

To get your ETH back from a Lockdrop User Contract, once your lock duration has ended, send a zero-value or greater transaction from any account to the contract address of the LUC.

The LUC will then return the ETH to the original address that sent this ETH to the Master Lockdrop Contract at the start of the Lockdrop Event. [Read a longer, detailed explanation at Blog.edgewa.re](#)

How can I access the EDG I got allocated to by lockdrop (including signaling)?

At the time of participating in lockdrop, you must have generated a mnemonic seed phrase. Kindly import it on Polkadot UI (optionally through Enzyme extension or polkadot(.js) extension) or on the unofficially maintained android version of the Math wallet and you will get access to your EDG! Read more about account interactions here:

<https://docs.edgewa.re/understanding-edgeware/accounts>

How many EDG will be minted in the genesis lockdrop event?

The genesis of the Edgeware network will mint 5 Billion EDG tokens.

Of this 5 Billion, 90%, or 4.5 billion will be distributed through the lockdrop process. The 10% remainder is distributed to:

4.5% to Commonwealth Labs, the developers of Edgeware, and will not be used to fund development. 3% to Parity Tech for support services on their Substrate product. 2.5% reserved for Community Incentives, OSS development, or other network success goals.

For the 90% distributed in the Lockdrop Event, lockdrop participants obtain 'shares' or the ability to obtain EDG at a certain proportion to their locked or signaled ETH. The following chart shows these share weights:

Duration of Lock	Method	Weighting	EDG Received
0 months	Signal	0.20x	25% at launch, 75% 265 days later
3 months	Lock	1.00x	Upon Network Launch
6 months	Lock	1.30x	Upon Network Launch
12 months	Lock	2.20x	Upon Network Launch

One's final shares are a proportion of the 90%, not a determinate value until the lockdrop closes and the total amount of locked ETH are known. One can earn greater share weights through locking for longer durations or locking earlier.

Lockdrop Early Participation Bonus: Schedule

Date Range (2019)	Bonus	Eth Cap
June 1 - June 15	S50%	No Cap
June 16 - June 30	35%	No Cap
July 1 - July 15	23%	No Cap
July 16 - July 30	14%	No Cap
July 31 - August 14	8%	No Cap
August 15 - August 29	5%	No Cap
August 30 - August 31	0%	No Cap, end of lockdrop

For <https://stats.edgewa.re>, we are planning to include a calculator with an estimation tool.

How can I view participation statistics and results from the lockdrop?

At Commonwealth.im: <https://commonwealth.im/#!/stats/edgeware>

I am signaling participant. When my EDG will become transferable?

25% EDG of signaling participants are already unlocked (transferable) and rest 75% EDG of are vested until 17th Feb 2021 (1 year from mainnet launch).

Why infinite supply / no max cap on supply/ inflation needed?

Inflation is necessary to reward those who secure the network (validators and nominators) and it is also utilized for ecosystem growth by means of the treasury. Currently, inflation is set to 95 EDG per block. No one can have direct access to those EDG. Approx 20% gets distributed as staking rewards and the rest goes to the treasury.

Why On-chain Treasury?

It's an on-chain fund which no one has direct access to! Edgeware has taken an incubator stance and seeking to support innovative projects/ideas utilizing the Edgeware network as a deployment platform. On-chain treasury provides funding(s) to such projects, onchain contractors and many innovative concepts on getting approval from the EDG holders directly or indirectly(by means of onchain council).

Why can't I see my balance after network upgrade?

Due to the network upgrade, some accounts need migration. Anyone can migrate anyone's account. More detailed information here:

<https://commonwealth.im/edgeware/proposal/discussion/625-edgeware-postupgrade-account-migration>

TechnicalFAQs

What is the difference between a node and a runtime upgrade? How do they interact?

Node upgrade is an update to the client software. These changes are mostly about networking software and/or optimizations to the system that processes the runtime binary.

While runtime upgrade is a change in the underlying WebAssembly binary being run by the node/client software. These are changes to the blockchain's state transition function.

New Users

General FAQ

Is there an Ambassador Program?

Edgeware has a set of Working Groups open to anyone who wants to participate - doocracy!

<https://commonwealth.im/edgeware/proposal/discussion/373-introducing-edgeware-working-groups>

How can I view participation statistics and results from the lockdrop?

At Commonwealth.im: <https://commonwealth.im/#!/stats/edgeware>

How do DOTs interact with EDG?

Initially, Edgeware is being launched as an independent chain ("solochain"). This means that EDG will be used as the bonding and reward token for validators.

When Polkadot launches, Edgeware will be eligible to become a parachain. DOTs are used to provide shared security and for inter-parachain communication in the Polkadot Network, so if and when Edgeware becomes a Polkadot parachain, for the duration of that parachain status, it will not have validators for the state of finality of its own , but rather that validation will happen through the Polkadot relaychain's validator pool. However, EDG will still be used for gas fees, spam prevention, and bonding for on-chain activities (e.g. governance).

In the future, the network may vote to make Edgeware a relay chain, just like Polkadot. In this case, EDG may be used to provide security for child parachains.

Are there video guides?

Here is a list of links for community videos and translated guides:



Lockdrop guides: videos and resources - (multi-language!)

<https://blog.edgeware.re/other-videos-guides-and-resources-multi-language-for-the-lockdrop/>

What will Commonwealth Lab's relation to Edgeware be after launch?

We will remain a core developer interested in the health and success of the network, but once Edgeware launches, Commonwealth Labs will participate in governance just like any other as a minority token holder. We're building a governance UI, anonymous voting modules, and other governance tech that we will deploy on Edgeware and other networks. We hope that our improvements are voted in by EDG holders and implemented on Edgeware!

What are the staking and token economics of the network?

There are initially 5,000,000,000 (five billion) EDG tokens minted, divisible up to 18 decimal places. Initially, inflation is set to 158 EDG per block. This implies approximately 997,220,160 EDG in the first year, or just under 20% inflation. The total amount of EDG minted will remain the same year after year, causing the percentage inflation to be disinflationary, with yearly inflation falling to approximately 16.6% in the second year. Additionally, a system-wide vote may further increase or decrease inflation. Half of the inflation will be voted upon by token holders for various uses.

How many validators will there be?

Edgeware is intended to have a very large validation community, ideally several thousand when the network is mature. Additionally, any individual can delegate (hence, DPoS) to a Validator.

Why should I validate?

Validators help provide security on Edgeware, since it's a PoS-based chain. Validators also earn fees from both the block reward and transactions. Earned tokens can be used to influence further governance decisions on the network. Validation is technical, and can be challenging. It may also be easier to delegate to a validator instead.

How does Edgeware utilize Parity Substrate?

Parity Substrate allows Edgeware developers to focus on improving the chain rather than developing infrastructure. Chains launched on Substrate generally do not have to deal with network or runtime level engineering changes. They can be natively extended with modules, which are written in Rust, compiled to Wasm, and linked into the client runtime. Modules can be voted into a chain by on-chain governance, at which point all clients will automatically download and run them, in a safe sandboxed environment. This makes the process of upgrading a chain much simpler and more accessible to a wide variety of developers!

What happens if Edgeware is no longer a parachain once Polkadot Network exists?

Even if Edgeware is voted off the Polkadot parachain set or our lease otherwise expires, it will still work as a "solochain", where it's responsible for its own security. In that case, EDG validators will have to reboot the chain with the chain state at the time Edgeware exited the Polkadot relay chain.

What will incentivize the dApp developers onto Edgeware over other networks?

There are numerous reasons to build on Edgeware. The first is that Substrate and the eventual Polkadot parachain status makes interoperability easy. Another is that the infrastructure is built with a light client-first mentality. This will enable future mobile portability to be seamless.

How does Edgeware benefit the Ethereum ecosystem?

We are Ethereum supporters! We also believe Ethereum will benefit from the growth of complementary ecosystems.

1. Lockdrop: The lockdrop will timelock ETH for up to a year as people go "long ETH and EDG". Remember, only ETH holders can participate! This adds another layer of utility to ETH and lowers circulating supply, which will likely enhance the economic security of Ethereum as it switches to a Casper-style consensus.

2. Bridging Networks: We think one of the first proposals that the community should work on is a game-theoretically sound ETH-EDG bridge. In the long term, this will allow fees and value to both to ETH and EDG. With a bridge in place, we anticipate that Ethereum dapps/protocols will be able to use Edgeware to scale, much in the same way that Loom Network and other protocols help scale Ethereum. Similarly, any tokens created on Edgeware can be custodied and "moved" over to Ethereum via the bridge, where they can be used in Ethereum's DeFi network (e.g. traded on 0x, collateralized on Maker and Compound, and bundled using Set).
3. Accelerating Development: Edgeware is a progressive network that will incorporate scaling and governance tech at a faster pace than Ethereum, but we expect most of these improvements will be backported to Ethereum. We think the net effect will be substantially beneficial for both ecosystems. As one example, at Commonwealth we are building a multi-chain governance tool to make governance on ETH, EDG, or any other chain easier, and Edgeware will allow us to test out several governance models that we will apply across the ecosystem.

Is Edgeware a fork of Ethereum?

No, it's a new chain built on a completely different codebase (Parity Substrate) with a different runtime and security model. While Ethereum holders can participate in the lockdrop, EDG will otherwise be an entirely separate network. However, we anticipate that a bridge will be built for ETH-EDG so that both chains can work together.

I don't see proof of my rewards from nominating.

Substrate is a little weird- there won't be a transaction or event that will show you where your balance increases from staking rewards. Using a Block Explorer that shows you your full balance like <https://edgeware.Subscan.io>, enter your reward-destination (stash or otherwise) account address and monitor the balance over a day to check that it is increasing.

How do I claim my EDG from the lockdrop?

No claim is necessary. When you participated in the lockdrop, you created an EDG Address. Your EDG is already at that address. You can check your balance with a Block Explorer. You may also need to convert your lockdrop address into the new format for a block explorer to find your account.

Why is my withdrawal from an exchange taking a long time?

Edgeware launched with an early version of Substrate that experiences a lag in block finality. Most exchanges wait until blocks are finalized to process withdrawals. Most users see these complete within a few hours. This will be remedied in the Summer 2020 Upgrade to Substrate 2.

FAQs for New Users

Accounts & Balances

Where can I store EDG?

For storing [and optionally staking], you can use the recommended full-fledged wallet interface [Polkadot Apps](#) [along with optional [polkadot{.js} extension](#)]. [Watch a walkthrough](#) □

[Clover extension wallet](#)(beta) can also be used as an alternative to the Polkadot JS extension.

If you are looking for mobile wallets, [Polkawallet](#) and [Math wallet](#) are the options that support EDG and certain web3 functionalities on Edgeware network.

Also, Ledger support is currently being developed by Zondax.

I can't find my account using my public address in a block explorer.

Most block explorers require the version of the Edgeware Public Address that encodes the Edgeware Network ID - if you created your key/address in the Lockdrop or via the Polkadot UI or extension, you first need to regenerate your address using a program called Subkey before you can use it to search. You can always use your seed to connect to wallet services, regardless of the network ID.

I'm seeing two different addresses depending on what service I use.

Edgeware accounts may have two address forms - one that encodes a Substrate Default Network ID and one that encodes the Edgeware network ID. Using block explorers, you should use the Edgeware form, and for all other cases we recommend the Edgeware network ID form, regenerate yours by following these instructions:

EDG Token

Where can I buy or acquire EDG? Is it listed?

At this time, there are several low liquidity markets (~150\$ daily volume) that list EDG. You should examine the security of these exchanges before making any purchases. Never buy seed phrases directly, your funds can be stolen.

Staking, Validation and Nomination

Can I start staking EDG?

Yes, Edgeware launched with a fully functional proof-of-stake nomination (also known as delegation,) and validation system. Any user that holds EDG can participate in securing the network and receiving benefits (but also endure the risks of being slashed for a validators nonconformance.)

How long does it take a waiting validator to become active? Why don't my nominators get selected?

Validators are elected from a pool of the top 60 most-bonded validators. This election method is not based on amount alone, but a complex algorithm called the Phragmen Method.

When can I unbond my nominated tokens?

Bonding periods last 7 days. After 7 days you need to click on the lock icon or use the withdrawUnbonded(num_slashing_spans) staking function [in Extrinsic submission option under Developer's tab] which will prompt you to sign a transaction to make those EDGs transferable.

Why can't I send/sign my Nomination transaction?

- Check that the Controller account has funds to pay transaction fees, this is the most common reason for issues. **If you get this error:** `staking.nominate submitAndWatchExtrinsic(extrinsic: Extrinsic): ExtrinsicStatus:: 1010: Invalid Transaction: Payment`
- Check that your Destination field in the Polkadot UI is set to the network default - and not to Kusama or other networks.

Why is it recommended to use a different account for Stash and Controller?

Separation of roles helps understand what accounts are performing what transactions, and allows us to disconnect the stash for security - making our wallets 'colder' while still authorizing a controller to perform certain actions. You dont have to do this, but it is recommended.

How often do I receive a reward from Staking?

Every 'era,' which is roughly 6 hours. An era is a category of block time. [See more about time.](#)

How do I see my bonded versus frozen versus free balances in my accounts?

You can use the Polkadot UI, then click Chain State tab to query your account for various parameters.

Polkadot UI

How do I connect to Edgeware Mainnet? OR I tried but it won't connect.

Click the network logo in the top right, and select Edgeware mainnet from the drop down. If you experience connection issues, try changing the endpoint using the custom endpoint option. See the following for other network endpoints:

Governance

One-person-one vote - what will prevent malicious players from signing on multiple nodes if it will cost little?

We provide a variety of tallying rules for certain governance features on Edgeware. The core governance is run with coin-weighted voting, with extensions that mimic a form of quadratic

voting based on locking time (i.e you get more weight by locking longer). Therefore what you ask is not wholly correct. Some tally types are inherently and unavoidably vulnerable to certain tactics- if someone wants to run a signaling poll with one person one vote that's their decision, but we can provide guidance about results and process.

Miscellaneous

How Edgeware is different is that 'x' project?

Every project is unique in its own perspective. You may find some similarities in Edgeware and the 'x' project. Though we should consider different projects as one of the essential parts of the ecosystem rather than non-healthy competitors. Also, it wouldn't be a good idea to talk on 'x' project's behalf. As Edgeware and the 'x' project offer different prospects to the Crypto Ecosystem by different means which you can't directly compare. There will be many blockchains with complex native smart contract support like Edgeware has. But Edgeware aims to be an ecosystem and one-stop solution platform for various types of deployments. Native EVM support, interoperability with the current DeFi ecosystem on Ethereum through edgeth bridges, incubator stance, initiatives through different working groups are the major proposed things which make Edgeware different. (Being a potential parachain also opens up possibilities not only for the polkadot relay chain but also for other parachains to utilize the bridges built by Edgeware.)

We would recommend going through the following resources to know more about Edgeware:

- [Roadmap](#)
- [Edgeware Docs](#)

Can I trade EDG on decentralised protocols like Uniswap or any DEXs?

Currently, you can't trade EDG on any swap protocols/portals like Uniswap. EDG is a native token of Edgeware blockchain, not an erc20 token. But there are plans for Uniswap integration through Tokyo Network Upgrade (Bridging to Ethereum) and it is scheduled for Fall of 2020. Edgeware aims to build Ethereum Bridges to setup interoperability of EDG into the existing ETH DeFi ecosystem, allowing for the creation of EDG / ETH, wEDG / DAI, and other pairs on Uniswap and Balancer.

Recently I participated in democracy voting. How can I make those locked EDGs

transferable after the locking period gets over?

Go to the Exinsics option under Developer tab in Polkadot Apps and by selecting 'democracy' under the extrinsic type, submit extrinsic unlock(target).

Parachain Status

Edgeware is currently a standalone chain, or solochain, but may become a parachain on a relay like Polkadot network.

How do DOTs interact with EDG?

Initially, Edgeware is being launched as an independent chain ("solochain"). This means that EDG will be used as the bonding and reward token for validators.

When Polkadot launches, Edgeware will be eligible to become a parachain. DOTs are used to provide shared security and for inter-parachain communication in the Polkadot Network, so if and when Edgeware becomes a Polkadot parachain, for the duration of that parachain status, it will not have validators for the state of the finality of its own, but rather that validation will happen through the Polkadot relaychain's validator pool. However, EDG will still be used for gas fees, spam prevention, and bonding for on-chain activities (e.g. governance).

In the future, the network may vote to make Edgeware a relay chain, just like Polkadot. In this case, EDG may be used to provide security for child parachains.

How Edgeware will get a parachain slot in Polkadot Ecosystem?

Parachain bonding I scheduled for the Fall of 2020 through the London upgrade. As of now the structure for incentives to DOT holders is not yet designed. But before the Polkadot bonding, we will see Edgeware Canary Network bonding to Kusama. So it will be a kind of simulation to parachain bonding to Polkadot. Moreover, the auction(s) is an optional stage for parachain slots as few slots will be offered to the contributors in the ecosystem. Crowdfunding a slot is yet another option.

Validator FAQs

description: An FAQ page for Staking and Nominating questions.

What is staking?

Staking allows EDG holders to participate in the security and availability of Edgeware by leveraging their tokens to validate. Validators who stake EDG, have an operational validator node, and behave honestly will get rewarded with EDG. Actors who misbehave or who are unavailable/offline will have a portion of their stake slashed as a penalty.

What are the annual returns for staking?

The exact number will vary depending on the amount of EDG staked. We'll update this as we know more.

What do I need to stake?

To become a validator, you need a computer with recently up-to-date specifications, a stable and fast internet connection, and EDG to stake. If you do not have EDG to stake, it is also possible to convince nominators to nominate you. Once you have acquired enough stake to make it into the validator set, you will start validating.

To become a nominator, you only need to have some EDG to stake.

What is nominating?

A nominator publishes a list of validator candidates that they trust, and puts down an amount of EDG at stake to support them with. If some of these candidates are elected as validators, they share with them the payments, or the sanctions, on a per-staked-EDG basis. Unlike validators, an unlimited number of parties can participate as nominators. As long as a nominator is diligent in their choice and only supports validator candidates with good security practices, their role carries low risk and provides a continuous source of revenue.

Can I nominate multiple validators?

Yes. Validators are selected via the Phragmen Method. You can think of this as a version of "approval voting" - you can approve zero, one, or multiple validators (although of course, if you do not nominate any validators, you are not nominating and thus will not receive any rewards).

For a more in-depth explanation of Phragmen, please see the [Polkadot Wiki Phragmen](#) page.

What is the maximum annual interest and rewards possible when nominating or validating?

Returns will vary based on several factors including, how many EDG are staked for a given validator, how much your proportion is in that stake, and how many validators are in the set at a given time.

Annual inflation of the EDG supply will not exceed 20%.

Rewards from validating is expected to be ~ 20% annually, assuming no slashing and remaining in the validator set the entire time. Note that only some of the rewards come from supply inflation; other rewards come from transaction fees, tips, and the like.

Maximum rewards are obtained when the percentage of all EDGs staked is at 80%.

What do I need to nominate?

All you need are some EDG and decide which validator to nominate.

Why should I validate on Edgeware?

Edgeware is a proof-of-stake network, which requires Validators to ensure transactions are well-formed and to provide security on Edgeware.

Validators also earn fees from both the block reward and transactions. Earned tokens can then be used to influence further governance decisions on the network.

I have 'x' active nomination(s), 'y' inactive nomination(s) and 'z' awaiting nomination(s) corresponding to my stash. What does it mean?

Active nominations indicate the nominations corresponding to which you are receiving the staking rewards. Ideally, all your nominations should be active, if not it means you are getting staking rewards partially for your stash.

Inactive nominations indicate your nominated validators which are currently offline. You need to stop nominations and renominate again to active validators to receive staking rewards corresponding to your whole stash. (Here, to stop nominations and renominate you don't need to unbond funds from your stash.)

Awaiting nominations are the ones which you recently nominated and they will be added to active nominations after a cycle of the era(6 hours).

What factors affect staking rewards?

Consider the following factors/parameters while staking:

- Lesser the commission, the higher your rewards will be.
- Nominating multiple validators will reduce the risk in case any slashing event occurs.
{So far no validator performed any malicious activity}
- Validators with a low total stake (self+nominated stake) will yield more reward per EDG.
But be sure your chosen validator has enough stake to maintain its position in the top 95 rankings by the total stake.

If you nominate to validator(s) having 100% commission (private validators) will get all your rewards and you won't receive any rewards.

Why can't I see any staking rewards?

Unless and until you choose the options of rewards to be paid in the stash account or controller account instead of the default option of bonding earned rewards, you won't see any transactions (on subscan) of rewards to your account.

If you had chosen the default option, you will see an increase in the bonded amount after claiming the payout(s) by you or anyone from the nominators or the validator.

How can one claim the staking rewards?

The staking rewards corresponding to a validator need to be claimed by a single transaction within 84 eras. Anyone from the validator or corresponding nominators can claim the rewards for everyone (top 256 nominators) by paying the transaction fees. If no one does it, rewards expire after 84 eras [84*(era length of 6 hours) = 504 hours i.e. 21 days]

Frequent rewards claiming could give better the compounding effect. However, beware that claiming payouts on behalf of all nominators and the validator could result in significant transaction fees.

Tools and Ecosystem

Chain Interfaces

- Commonwealth
 - Polkadot UI
-

Polkadot UI

The [Polkadot UI](#) is an essential interface for performing a variety of actions on Edgeware and Substrate-based chains. It will be the primary way that you interact with the Edgeware network, whether that is staking, account transactions, some governance functions, or more.

Secondary to the Polkadot UI is [Commonwealth.im](#), which combines discussion and polling on-chain with governance and account functions. As Commonwealth is still under development, many instructions on this site will be written for Polkadot UI.

Features

- Manage accounts, view account info.
- Send and receive/monitor transactions
- View chain, block and node info.
- Conduct staking, democracy, council and treasury actions.
- View chain state and parameters.
- Observe extrinsics
- Upload contracts

Wallets

- **Commonwealth.im (Sign-up in right top side, must connect EDG address)**
 - At this time, **the wallet is under development**. It supports viewing balances, sending and receiving transactions, and governance actions via the Commonwealth.im UI. The public addresses shown do not display the Edgeware network ID-encoded

version at this time, but this feature is due by end-of-launch-week. You should regenerate your public address using the link below.

- **Polkadot JS Browser Extension (Chrome and Firefox)**

- Can be used to connect EDG addresses to Commonwealth.im. (This is the Official Polkadot Wallet Extension)
- Supports the most features via [the Polkadot UI](#).
- The Polkadot.js Browser Extension **does not display the Edgeware network ID encoded at this time**. The public address of your EDG wallet shown in the extension is encoded using the 'default network ID' of Subkey, the program that generates keypairs for Substrate-based chains. As a result, the public addresses shown may not be useful for using block explorers. You should regenerate your public key to derive the Edgeware network-ID-encoded public address, see above.

- [MathWallet Browser Extension](#)

- [Polkawallet](#) - A mobile wallet for Polkadot on both iOS and Android. Currently in development but a Beta version is available for download. Follow development on [GitHub](#).

- [Enzyme](#) - Browser extension wallet. Follow development on [GitHub](#).

- [Sakura](#) - Desktop Wallet for Polkadot Ecosystem. In development.

- [Signer](#)

- [Lunie](#)

- [Trust Wallet](#)

- [ImToken](#)

- [Ownbit](#)

- [AirGap](#)

- [SafePal](#)

- [Crypto.com](#)

- [Ledger App](#)

- [Atomic Wallet](#)

- [Dether](#)

- [Cobo Wallet](#)

- [Swipe](#)

- [Polkadot{.js} extension](#)

- [MetaMask](#)

- [Speckle](#)

- [KodaDot](#)

- [Subwallet](#)

- [Fearless](#)

- [Spatium](#)

- Blockchain.com

Exchanges

 This list is provided without endorsement by Commonwealth Labs, the core developers of Edgeware. Using an exchange is at **your own risk**, and you should investigate fully the risks and the character of the entity. **You can also check CoinGecko for listings.**

Exchange	Pairs	URL
HotBit	EDG/BTC	https://www.hotbit.io/exchange
MXC	EDG/USDT	https://www.mxc.com/trade/easy#EDG_USDT
BKEX	EDG/USDT	https://www.bkex.com/trade/EDG_USDT
HotBit	EDG/USDT	https://www.hotbit.io/exchange?symbol=EDG_USDT
MXC	EDG/BTC	https://www.mxc.com/trade/easy#EDG_BTC
Biki	EDG/USDT	https://www.biki.com/en_US/trade/EDG_USDT
BiHodl	EDG/ USDT	https://www.bihodl.com/#/exchange/edg_usdt

Nodes and Deployment

- Gantree Infrastructure Toolkit for Substrate
- Polkalert: Node Monitoring
- Telemetry: Node Stats

Block Explorers

- Polkadot-JS Apps Explorer - Polkadot dashboard block explorer. Currently connects to Kusama by default, but can be configured to connect to other

remote or local endpoints.

- [Polkascan](#) - Blockchain explorer for Polkadot, Kusama, and other related chains. [Repo](#).
- [Subscan](#) - Blockchain explorer for Substrate chains. [Repo](#).

WASM

Webassembly related tools and projects.

- [ink!](#) - an eDSL to write WebAssembly based smart contracts using the Rust programming language.
- [parity-wasm](#) - Low-level WebAssembly format library.
- [wasm-utils](#) - Collection of WASM utilities used in pwasm-ethereum and substrate contract development.
- [wasmi](#) - a Wasm interpreter conceived as a component of parity-ethereum (ethereum-like contracts in wasm) and substrate.

Smart Contract tooling

- [Halva](#) - a Truffle-inspired local development environment for Substrate.
- [Redspot](#)

Faucets

- [BlockX Labs testEDG Faucet](#)

Grant Programs

- [Edgeware Grant Program](#)
- [Web3Foundation Grant Program](#)

API Feeds

- [Total Supply Endpoint for Edgeware](#) (Does not subtract Treasury stash)

Key Generation and Conversion

- Original Edgeware Key Generator for Lockdrop and Pub Key Convertor
- Subkey: Key Generation for Substrate Chains

Node-as-a-Service Providers

These providers have not been vetted by the core development team.

Name	Link to Product
BisonTrails	https://bisontrails.co/edgeware/

The following have announced they intend to develop NAAS.

- Ankr.com
- Stake.Fish
- <https://deploy.radar.tech/>

Network Monitoring & Reporting

- **Polkadot Telemetry Service** - Network information including what nodes are running the chain, what software versions they are running, sync status, and location.
- Polkabot - Polkadot network monitoring and reporting using Matrix (Riot / Element) chat. Users may create custom bot plugins. [Blogpost](#).
- **Ryabina's Telegram Bot** - a Telegram bot for monitoring on-chain events of Substrate chains.
[Github Repository](#)
- **PolkaStats** - Polkadot network statistics (includes Kusama). Shows network information and staking details from validators and intentions.
[Github Repository](#).
- **Panic** - a node monitoring and alerting server for validators.

- [OpenWeb3/Guardian](#) - a CLI tool & JS library to monitor on chian states and events.
-

Clients

- [Polkadot](#) - Rust implementation of the Polkadot Host.
 - [Kagome](#) - A C++ Polkadot client developed by [Soramitsu](#).
 - [Gossamer](#) - A Go implementation of the Polkadot Host.
 - [Polkadot-JS client](#) - Alternative client for JavaScript enthusiasts.
 - [TX Wrapper](#) - Helper funtions for offline transaction generation.
-

Tools

- [Substrate](#) - Blockchain development platform written in Rust. Polkadot is being built on top of Substrate.
- [Substrate Knowledge Base](#) - Comprehensive documentation and tutorials for building a blockchain using Substrate.
- [Substrate VSCode plugin](#).
- [Substrate Debug Kit](#) - A collection of debug tools and libraries around substrate chains. Includes tools to calculate NPoS elections offline, disk usage monitoring, test templates against chain state and other pallet-specific helper.
- [Diener](#) - a tool for easy changing of Polkadot or Substrate dependency versions.
- [Polkadot Launch](#) - a tool to easily launch custom local parachain-enabled Polkadot versions.
- [Halva](#) - a Truffle-inspired local development environment for Substrate.
- [Fork-off Substrate](#) - copies the state of an

existing chain into your local version and lets you further experiment on it.

- [srtool](#) - a tool for verifying runtime versions against on-chain proposal hashes.
 - [sub-bench](#) - a tool to spam your node with transactions for the sake of benchmarking.
 - [substrate-devhub-utils](#) - a set of JavaScript utilities making life with Substrate a little easier.
-

UI

- [Polkadash](#) - VueJS-based starter kit for custom user interfaces for Substrate chains. [Tutorials](#).
 - [Polkadot JS Apps UI](#) - Repository of the polkadot.js.org/apps UI.
 - [Substrate Front-end Template](#) - ReactJS-based starter UI for custom user interfaces for Substrate chains.
 - [Polkadot JS Browser Extension](#) - key management in a Chrome extension.
-

Libraries

Polkadot-JS Common

Polkadot-JS Common provides various utility functions that are used across all projects in the `@polkadot` namespace and is split into a number of internal utility packages. The documentation and usage instructions are provided at [Polkadot-JS/Common API Documentation](#).

- [@polkadot/keyring](#) To create / load accounts in JavaScript, helpful for creating wallets or any application that will require the user to write to chain. [Examples](#).
- [@polkadot/util](#) Utility functions like checking if a string

- is hex-encoded.
- [@polkadot/util-crypto](#) Crypto utilities that will come in handy while developing with Polkadot.

CLI Tools

- [@polkadot/api-cli](#) Command line interface for the polkadot API. [Documentation](#).
- [@polkadot/monitor-rpc](#) RPC monitor for Polkadot. See the RPC tools below for additional information.
- [@polkadot/signer-cli](#) Tool to construct, sign, and broadcast transactions. Signing can be done offline.
- [Polkadot API Cpp](#) - C++ API for Polkadot, can build `clip`, a command line tool.

WASM

Webassembly related tools and projects.

- [ink!](#) - an eDSL to write WebAssembly based smart contracts using the Rust programming language.
- [parity-wasm](#) - Low-level WebAssembly format library.
- [wasm-utils](#) - Collection of WASM utilities used in pwasm-ethereum and substrate contract development.
- [wasmi](#) - a Wasm interpreter conceived as a component of parity-ethereum (ethereum-like contracts in wasm) and substrate.

RPC and API Tools

- [@polkadot/api/rpc-provider](#) Demonstrates how the JS tools interact with the node over RPC.
- [RPC documentation](#).
- [Polkadot API Server by SimplyVC](#).
- [Go: Subscan API](#).
- [C++ Polkadot API](#) - C++ API for Polkadot.
- [.NET Polkadot API](#) - Polkadot Substrate API for .NET.

- [Python Polkadot API](#).
- [GSRPC](#) - Substrate RPC client in Go,
a.k.a. GSRPC.
- [Substrate API Sidecar](#) - an HTTP wrapper for
Substrate, abstracting some complex RPC calls into simple REST calls.
- [Subxt](#) - a Rust library to submit extrinsics to a
Substrate node via RPC.

SCALE Codec

The [SCALE](#) (Simple Concatenated Aggregate Little-Endian) Codec is a lightweight, efficient, binary serialization and deserialization codec.

It is designed for high-performance, copy-free encoding and decoding of data in resource-constrained execution contexts, like the Substrate runtime. It is not self-describing in any way and assumes the decoding context has all type knowledge about the encoded data.

It is used in almost all communication to/from Substrate nodes, so implementations in different languages exist:

- [Ruby](#)
- [Rust](#)
- [Go](#)
- [C++](#)
- [TypeScript](#)
- [AssemblyScript](#)
- [Haskell](#)
- [Java](#)
- [Python](#)

Data Crawling and Conversion

The following tools help you extract and structure data from a Substrate node.

- [Polkascan PRE Harvester](#)

(matching explorer for harvested data).

- Parity's Substrate Archive.
- Hydra: GraphQL Builder.
- Polka-store - a tool which scans a Substrate chain and stores balance-relevant transactions in an SQLite database.
- Substrate-graph - A compact indexer for Substrate based nodes providing a GraphQL interface.

Miscellaneous

- GSRPC: Substrate RPC Client in Go
- Chronologic Transaction Scheduler
- Edgeware Secure Mnemonic Phrase Generator for Lockdrop

Networks and Public Endpoints

Publically Available Nodes on Mainnet and Testnets

Explore these networks with a Block Explorer.

- `wss://networkNameNumber.edgewa.re` or (if `wss:` fails)
- `ws://networkNameNumber.edgewa.re:9944`

Edgeware Mainnet

At this time, we recommend using the node run by Patract Labs Elara or the OnFinality node to balance load.

Patract Elara Public Node

```
wss://edgeware.elara.patract.io
```

OnFinality Public Node

```
wss://edgeware.api.onfinality.io
```

Commonwealth Labs Nodes

```
1 wss://mainnet1.edgewa.re
2 wss://mainnet2.edgewa.re
3 wss://mainnet3.edgewa.re
4 wss://mainnet4.edgewa.re
5 wss://mainnet5.edgewa.re
6 wss://mainnet6.edgewa.re
7 wss://mainnet7.edgewa.re
8 wss://mainnet8.edgewa.re
9 wss://mainnet9.edgewa.re
10 wss://mainnet10.edgewa.re
```

```
11 wss://mainnet11.edgewa.re
12 wss://mainnet12.edgewa.re
13 wss://mainnet13.edgewa.re
14 wss://mainnet14.edgewa.re
15 wss://mainnet15.edgewa.re
16 wss://mainnet16.edgewa.re
17 wss://mainnet17.edgewa.re
18 wss://mainnet18.edgewa.re
19 wss://mainnet19.edgewa.re
20 wss://mainnet20.edgewa.re
```

Edgeware Beresheet Testnet

Status Update (May 21, 2021) Confirmed that Beresheet1, Beresheet3 are working. Some Beresheet networks may have a javascript library issue that is not updated.

```
1 wss://beresheet1.edgewa.re
2 wss://beresheet2.edgewa.re
3 wss://beresheet3.edgewa.re
4 wss://beresheet4.edgewa.re
5 wss://beresheet5.edgewa.re
6 wss://beresheet6.edgewa.re
7 wss://beresheet7.edgewa.re
8 wss://beresheet8.edgewa.re
9 wss://beresheet9.edgewa.re
10 wss://beresheet10.edgewa.re
```

Canarynet Functionality

Coming soon.

Exchanges

 This list is provided without endorsement by Commonwealth Labs, developers of Edgeware. Using an exchange is at your own risk, and you use should investigate fully the risks and the character of the entity. **You can also check CoinGecko for listings.******

Exchange	Pairs	URL	⚠️ Warning
BiHodl	EDG/ USDT	https://www.bihodl.com/#/exchange/edg_usdt	
BKEX	EDG/USDT	https://www.bkex.com/trade/EDG_USDT	
MXC	DG USDT	https://www.mxc.com/trade/easy#EDG_USDT	May be staking or voting with users tokens.
HotBit	EDG/BTC and EDG/USDT	https://www.hotbit.io/exchange	
Biki	EDG/USDT	https://www.biki.com/en_US/trade/EDG_USDT	Has been known to disable withdrawal of EDG involuntarily.

Gate.io EDG/USDT https://www.gate.io/en/trade/EDG_USDT

Gate.io
wallets have
been
suspended
after
transfers for
weeks.

Style Guide

Documentation Style Guide

UI Support Section

There are many user-executable functions in Edgeware. It is helpful to note what interfaces support functions. Insert a single column table-list with hyperlinked names of the interfaces that support the function.

Example:

Function Supported By:

Polkadot UI

Commonwealth UI

Polkascan

Polkassembly

Linking to Substrate Documentation

Where possible, insert a hint with the link to the appropriate Substrate.dev documentation page. Bold the text "Substrate Documentation" and insert a link.

Example:

 **Substrate Documentation** for Treasury Module

Parameter Noting

Whenever documentation refers to a parameter modifiable through governance, a "hint" module should be used to highlight that parameter, describe in short the function, and assert the current parameter status. It should also be updated in the master parameter list. Bold the text "Parameter Note:"

Example



Parameter Note: This hint block should name the parameter and state the current setting.

Code Style Guide

Formatting

- Indent using tabs.
- Lines should be longer than 100 characters long only in exceptional circumstances and certainly no longer than 120. For this purpose, tabs are considered 4 characters wide.
- Indent levels should be greater than 5 only in exceptional circumstances and certainly no greater than 8. If they are greater than 5, then consider using `let` or auxiliary functions in order to strip out complex inline expressions.
- Never have spaces on a line prior to a non-whitespace character
- Follow-on lines are only ever a single indent from the original line.

```
1 fn calculation(some_long_variable_a: i8, some_long_variable_b: i8) -> bool
2     let x = some_long_variable_a * some_long_variable_b
3         - some_long_variable_b / some_long_variable_a
4             + sqrt(some_long_variable_a) - sqrt(some_long_variable_b);
5     x > 10
6 }
```

- Indent level should follow open parens/brackets, but should be collapsed to the smallest number of levels actually used:

```
1 fn calculate(
2     some_long_variable_a: f32,
3     some_long_variable_b: f32,
4     some_long_variable_c: f32,
5 ) -> f32 {
6     (-some_long_variable_b + sqrt(
7         // two parens open, but since we open & close them both on the
8         // same line, only one indent level is used
9         some_long_variable_b * some_long_variable_b
10            - 4 * some_long_variable_a * some_long_variable_c
11        // both closed here at beginning of line, so back to the original indent
12        // level
13    )) / (2 * some_long_variable_a)
14 }
```

- `where` is indented, and its items are indented one further.
- Argument lists or function invocations that are too long to fit on one line are indented similarly to code blocks, and once one param is indented in such a way, all others should be, too. Run-on parameter lists are also acceptable for single-line run-ons of basic function calls.

```

1 // OK
2 fn foo(
3     really_long_parameter_name_1: SomeLongTypeName,
4     really_long_parameter_name_2: SomeLongTypeName,
5     shrt_nm_1: u8,
6     shrt_nm_2: u8,
7 ) {
8     ...
9 }
10
11 // NOT OK
12 fn foo(really_long_parameter_name_1: SomeLongTypeName, really_long_parameter_name_2: SomeLongTypeName, shrt_nm_1: u8, shrt_nm_2: u8) {
13     ...
14 }
15 }
```

```

1 {
2     // Complex line (not just a function call, also a let statement). Full
3     // structure.
4     let (a, b) = bar(
5         really_long_parameter_name_1,
6         really_long_parameter_name_2,
7         shrt_nm_1,
8         shrt_nm_2,
9     );
10
11     // Long, simple function call.
12     waz(
13         really_long_parameter_name_1,
14         really_long_parameter_name_2,
15         shrt_nm_1,
16         shrt_nm_2,
17     );
18
19     // Short function call. Inline.
20     baz(a, b);
21 }
```

- Always end last item of a multi-line comma-delimited set with `,` when legal:

```

1 struct Point<T> {
2     x: T,
3     y: T,    // <-- Multiline comma-delimited lists end with a trailing ,
4 }
5
6 // Single line comma-delimited items do not have a trailing `,`
7 enum Meal { Breakfast, Lunch, Dinner };

```

- Avoid trailing `;`s where unneeded.

```

1 if condition {
2     return 1    // <-- no ; here
3 }

```

- `match` arms may be either blocks or have a trailing `,`, but not both.
- Blocks should not be used unnecessarily.

```

1 match meal {
2     Meal::Breakfast => "eggs",
3     Meal::Lunch => { check_diet(); recipe() },
4 //     Meal::Dinner => { return Err("Fasting") }    // WRONG
5     Meal::Dinner => return Err("Fasting"),
6 }

```

Style

- Panickers require explicit proofs they don't trigger. Calling `unwrap` is discouraged. The exception to this rule is test code. Avoiding panickers by restructuring code is preferred if feasible.

```
1 let mut target_path =  
2     self.path().expect(  
3         "self is instance of DiskDirectory;\  
4             DiskDirectory always returns path;\  
5             qed"  
6 );
```

- Unsafe code requires explicit proofs just as panickers do. When introducing unsafe code, consider tradeoffs between efficiency on one hand and reliability, maintenance costs, and security on the other.

Here is a list of questions that may help evaluating the tradeoff while preparing or reviewing a PR:

- how much more performant or compact the resulting code will be using unsafe code,
- how likely is it that invariants could be violated,
- are issues stemming from the use of unsafe code caught by existing tests/tooling,
- what are the consequences if the problems slip into production.