blockchained
{'india'}

presents

60 Days of Code with

Polkadot

Instructions:

1. Ensure that the task is completed on the very same day it is assigned. If you face any problem, do share it with us in the group at the same moment, so that the doubts are cleared at the earliest.

2. Every week we will complete one module. Each module is further divided into 5 days. We have kept the weekends free so you can catch up on the work over the weekend and attend the doubts session/workshops which will be organised.

3. If you are not able to complete the task, ensure that they are covered in the very week so that the process is not delayed much.

4. We will have live workshops every week. The schedule of the same will be shared at the start of the week and the same is added to your weekly curriculum. Do ensure you are attending these workshops as they are live sessions by industry experts. It is going to help you only in your journey. If by any reason you are not able to attend, inform the team beforehand.

5. Please submit your assignment as these will be necessary to participate in the hiring track.

6. Please update your daily progress by posting on twitter and sending a message on the group When posting on your twitter accounts please tag us @blockchainedind and @Polkadot, along with the use of following hashtags:
#60DaysofCode #BuildInPublic #Polkadotdevbootcamp

If you have any questions or doubts, please reach out to us on the group or email us at hriday@blocumen.com and manav@blocumen.com.

Content

Module 4: Building Blockchain with substrate
(June 25 to July 1)

Module 4 : Building Blockchain with substrate

Day 1

Building a Local Blockchain

1.Building a local blockchain with Substrate:

Getting Started

Welcome to the captivating world of Substrate and Polkadot blockchain development! Are you ready to take the first step towards becoming a skilled blockchain developer? In this module, we will embark on a thrilling journey as we learn how to compile and launch a single local blockchain node.

Throughout this module, you will gain hands-on experience in building and starting your very own blockchain using the node template. The Substrate node template is an incredible tool that provides a fully functional single-node blockchain, right at your fingertips. You can easily run this blockchain locally in your development environment, empowering you to experiment and perform a multitude of tasks.

Even without making any changes to the template, you'll be able to run a fully functional node that not only produces blocks but also enables seamless transaction processing. It comes equipped with various preconfigured components like user accounts and account balances, allowing you to dive straight into exploring the blockchain's capabilities.

But that's not all! Once you have successfully launched your local blockchain node, this module takes it up a notch by introducing you to a Substrate front-end template. With this powerful tool, you'll be able to witness and analyze real-time information

about blockchain activities. Moreover, you'll learn how to submit transactions, taking your interaction with the blockchain to a whole new level.

Get ready to unleash your creativity and discover the limitless possibilities of Substrate and Polkadot blockchain development. Let's dive in and embark on this transformative learning journey together in this captivating module!

This module serves as an introductory guide to building a local blockchain with Substrate. It establishes a fundamental understanding of Substrate and facilitates the setup of a minimal working development environment, enabling you to delve deeper into subsequent modules. Designed for individuals eager to explore Substrate and engage in blockchain development, this module assumes no prior experience or knowledge of Substrate.

Prerequisites for this module

1. Stable Internet Connection: Ensure that you have a reliable internet connection throughout the module, as you'll need to download necessary dependencies and access online resources.

2. Shell Terminal Access: Familiarity with using a shell terminal on your local computer is essential. You should be comfortable navigating directories, running commands, and managing files through the command-line interface.

3. Software Development Knowledge: Having a general understanding of software development principles and practices will be beneficial. This includes concepts such as variables, functions, data structures, and programming logic.

4. Familiarity with Blockchains and Smart Contract Platforms: It's helpful to have a basic understanding of blockchains and smart contract platforms. Familiarize yourself with fundamental concepts like decentralized networks, consensus mechanisms, and the purpose of smart contracts.

5. Rust Installation and Development Environment Setup: Ensure that you have Rust installed and your development environment properly configured. This includes setting up the Rust toolchain, Cargo (Rust's package manager), and other necessary components for building Substrate-based projects.

By meeting these prerequisites, you'll be well-prepared to dive into the module and maximize your learning experience. If you need assistance with any of these prerequisites, there are resources available online to guide you through the installation and setup processes.

https://docs.substrate.io/install/

Objectives for this module

Upon finishing this module, you will achieve the following outcomes:

1. Successful Compilation and Launch of a Local Substrate Blockchain: You will have the knowledge and hands-on experience to compile and launch a local Substrate-based blockchain using the node template. This will enable you to have a functioning blockchain network running on your local development environment.

2. Installation of a Front-End Template for Interacting with the Blockchain: You will learn how to install a front-end template that allows you to interact with the local blockchain node. This front-end interface provides a user-friendly way to view and analyze real-time information about blockchain activities.

3. Transaction Submission and Result Viewing: Using the front-end template, you will understand how to submit transactions to the local blockchain node. You will gain practical experience in submitting transactions and be able to witness the results of those transactions.

By accomplishing these objectives, you will have a solid foundation in building and interacting with Substrate-based blockchains. This knowledge will empower you to continue exploring more advanced topics and take your blockchain development skills to the next level. You will be equipped with the necessary tools and understanding to participate in the exciting world of Substrate and Polkadot blockchain development.

Compile the Substrate Node Template

To start building on Substrate, you need to compile the Substrate node template, which provides a working development environment. Follow the instructions below to compile the Substrate node template:

1. Open a terminal shell on your computer.

2. Clone the node template repository by running the following command:

```
git clone
https://github.com/substrate-developer-hub/substrate-node-template
```

By default, this command clones the main branch of the repository. However, if you prefer to work with a specific Polkadot version and are familiar with it, you can use the `--branch` command-line option to select a specific branch. You can find the available Polkadot branches by clicking on "Releases" or "Tags" in the repository.

3. Change to the root of the node template directory by running the following command:

```
cd substrate-node-template
```

4. Create a new branch to contain your work. This step helps you keep track of your changes and isolate them from the main branch. Use the following command:

```
git switch -c my-learning-branch-yyyy-mm-dd
```

Replace `yyyy-mm-dd` with any identifying information that you desire. We recommend using a numerical year-month-day format. For example:

```
git switch -c my-learning-branch-2023-03-01
```

5. Compile the node template by running the following command:

```
cargo build --release
```

It's important to always use the `--release` flag when building to generate optimized artefacts. Please note that the initial compilation may take some time to complete.

Once the compilation is finished, you should see a line similar to the following:

```
Finished release [optimized] target(s) in 11m 23s
```

With the successful completion of the compilation process, you are now ready to proceed to the next steps and start working with your local Substrate node.

Running the substrate node

Now that your Substrate node has been compiled, it's time to start exploring its functionalities using the front-end template. Follow the instructions below to start the local Substrate node:

1. In the same terminal where you compiled your node, start the node in development mode by running the following command:

```
./target/release/node-template --dev
```

The `node-template` command-line options specify the desired operation mode for the running node. In this case, the `--dev` option indicates that the node should run in development mode using the predefined development chain specification. By default, this option also deletes all active data, including keys, the blockchain database, and networking information, when you stop the node using Control-C. Using the `--dev` option ensures a clean working state each time you stop and restart the node.

2. Verify that your node has started successfully by reviewing the output displayed in the terminal.

The terminal should display output similar to the following:

```
2022-08-16 13:43:58 Substrate Node
2022-08-16 13:43:58 ✌️  version 4.0.0-dev-de262935ede
2022-08-16 13:43:58 ❤️  by Substrate DevHub
<https://github.com/substrate-developer-hub>, 2017-2022
2022-08-16 13:43:58 📋 Chain specification: Development
2022-08-16 13:43:58 🏷  Node name: limping-oatmeal-7460
2022-08-16 13:43:58 👤 Role: AUTHORITY
2022-08-16 13:43:58 💾 Database: RocksDb at
/var/folders/2_/g86ns85j5l7fdnl621ptzn500000gn/T/substrate95LPvM/c
hains/dev/db/full
2022-08-16 13:43:58 ⛓  Native runtime: node-template-100
(node-template-1.tx1.au1)
2022-08-16 13:43:58 🔨 Initializing Genesis block/state (state:
0xf6f5...423f, header-hash: 0xc665...cf6a)
2022-08-16 13:43:58 👴 Loading GRANDPA authority set from genesis
on what appears to be first startup.
```

```
2022-08-16 13:43:59 Using default protocol ID "sup" because none
is configured in the chain specs
2022-08-16 13:43:59 🏷  Local node identity is:
12D3KooWCu9uPCYZVsayaCKLdZLF8CmqiHkX2wHsAwSYVc2CxmiE
...
...
...
...
2022-08-16 13:54:26 💤 Idle (0 peers), best: #3 (0xcdac...26e5),
finalized #1 (0x107c...9bae), ⬇ 0 ⬆ 0
```

Note: The log output shown here represents a sample log, and the displayed information may vary based on your specific setup.

3. If the number following "finalized" is increasing, it indicates that your blockchain is successfully producing new blocks and reaching consensus on the state they describe.

It's important to keep the terminal that displays the node output open as we will be utilizing it in the upcoming steps.

Congratulations! Your Substrate node is now up and running, actively producing blocks and functioning

Installing the Front end Template

The front-end template complements the Substrate node by providing a web browser interface powered by ReactJS. It enables you to interact with the Substrate-based blockchain node and serves as a starting point for creating user interfaces for your own projects. Follow the instructions below to install and run the front-end template.

1. Open a new terminal window on your computer.

2. Check if Node.js is installed by running the following command:

```
node --version
```

If Node.js is installed, it will display a version number like `v14.7.0`.

If the command doesn't return a version number, download and install Node.js by following the instructions provided on the Node.js website. Alternatively, you can install and use the nvm (Node Version Manager) to manage your Node.js installation.

Note: The front-end template requires at least Node.js version 14 to run.

3. Check if Yarn is installed by running the following command:

```
yarn --version
```

The version of Yarn should be at least `v3` to run the front-end template. If you have an older version of Yarn installed, you can update it by running `yarn set version <desired-version>`. If Yarn is not installed or you have any questions about installing a particular version, refer to the Yarn website for installation instructions.

Cloning and Installing the Front-end Template:

4. Clone the front-end template repository by running the following command:

```
git clone
https://github.com/substrate-developer-hub/substrate-front-end-tem
plate
```

5. Change to the root directory of the front-end template by running the following command:

```
cd substrate-front-end-template
```

6. Install the dependencies for the front-end template by running the following command:

```
yarn install
```

Running the front end template

7. Verify that your current working directory is the root directory where you installed the front-end template in the previous step.

8. Start the front-end template by running the following command:

```
yarn start
```

Typically, running the `yarn start` command automatically opens `http://localhost:8000` in your default browser. If the browser doesn't open automatically, you can manually enter `http://localhost:8000` in your browser's address bar to view the front-end template.

The top section of the front-end template displays an account selection list, allowing you to choose the account for on-chain operations. It also provides information about the connected chain.

Note: The front-end template includes a Balances table with predefined accounts and their associated balances. You can use this sample data to perform operations such as transferring funds.

Transferring funds to an Account

9. Locate an account with zero funds in the Balances table.

10. Below the Balances table, you'll find a Transfer component. Use this component to transfer funds from one account to another.

11. Select the account (e.g., "dave") to which you want to transfer funds.

12. Specify the amount you want to transfer (e.g., at least 1000000000000).

13. Click the "Submit" button to initiate the transfer.

14. Observe that the balances in the Balances table are updated to reflect the transfer.

15. Check the Events component to view events related to the completed transfer. The Events component displays details about each operation performed during the transfer.

16. Once the transaction is completed and included in a block, a confirmation message will be displayed.

stop the local node

17. If you wish to stop the local Substrate node and reset any state changes you've made,

return to the terminal where the node output is displayed.

18. Press `Control-C` to terminate the running process.

19. Verify that your terminal returns to the prompt in the `substrate-node-template` directory.

Congratulations! You have successfully installed and run the front-end template, interacted with the Substrate-based blockchain node, and performed a transfer operation.

Day 2

Simulating a local blockchain network

Substrate :

Introduction

In the previous day's module, we learned how to build a local blockchain using the Substrate node template. We explored the authority consensus model and started a blockchain node using a predefined account. Now, we will proceed to simulate a network by adding a second node to the existing blockchain network.

Module Objectives

By the end of this tutorial, you will achieve the following objectives:
1. Start a second blockchain node using a different account.
2. Understand the key command-line options for starting a node.
3. Verify that blocks are being produced and finalized in the network.

Instructions

1. Start the First Blockchain Node:

- Open a terminal shell on your computer.
- Change to the root directory where you compiled the Substrate node template.
- Purge old chain data by running the following command:

```
    ./target/release/node-template purge-chain --base-path
/tmp/alice --chain local
```

  Confirm the operation by typing 'y' when prompted.
- Start the local blockchain node using the alice account:

```
./target/release/node-template \
    --base-path /tmp/alice \
    --chain local \
    --alice \
    --port 30333 \
    --ws-port 9945 \
    --rpc-port 9933 \
    --node-key
```

```
0000000000000000000000000000000000000000000000000000000000000001 \
     --telemetry-url "wss://telemetry.polkadot.io/submit/ 0" \
     --validator
```

2. Add a Second Node to the Blockchain Network:

- Open a new terminal shell on your computer.
- Change to the root directory where you compiled the Substrate node template.
- Purge old chain data for the second node:

```
     ./target/release/node-template purge-chain --base-path
/tmp/bob --chain local -y
```

- Start a second local blockchain node using the bob account:

```
     ./target/release/node-template \
     --base-path /tmp/bob \
     --chain local \
     --bob \
     --port 30334 \
     --ws-port 9946 \
     --rpc-port 9934 \
     --telemetry-url "wss://telemetry.polkadot.io/submit/ 0" \
     --validator \
     --bootnodes
/ip4/127.0.0.1/tcp/30333/p2p/12D3KooWEyoppNCUx8Yx66oV9fJnriXwCcXwD
DUA2kj6vnc6iDEp
```

Note the differences in options between the first and second node startup commands.

3. Verify Blocks are Produced and Finalized:

- In the terminal where you started the first node, verify that you see the following messages indicating block production:

```
2022-08-16 15:32:33 discovered:
12D3KooWBCbmQovz78Hq7MzPxdx9d1gZzXMsn6HtWj29bW51YUKB
/ip4/127.0.0.1/tcp/30334
     2022-08-16 15:32:36 🙌 Starting consensus session on top of
```

```
parent
0x2cdce15d31548063e89e10bd201faa63c623023bbc320346b9580ed3c40fa07f
     2022-08-16 15:32:36 Received proof-of-work solution outdated
for round 2, solution will be ignored.
     2022-08-16 15:32:38 Imported #1 (0x0c81...080e)
     2022-08-16 15:32:38 🎁 Prepared block for proposing at 2
[hash: 0xe45a...ac8a; parent_hash: 0x2cdce1...fa07f; extrinsics
(1): [0x7b2e...e0c2]]
     2022-08-16 15:32:38 Pre-sealed block for proposal at 2. Hash
now 0xa4e0...1b0f
     2022-08-16 15:32:38 Imported #2 (0xa4e0...1b0f)
     2022-08-16 15:32:38 🔒 Imported finalized block #2
(0xa4e0...1b0f)
```

- The messages indicate that blocks are being produced and imported into the network. The finalized block is also indicated, which means consensus has been reached on that block.

- Leave the first terminal running to keep the first node active.

4. Verify Connectivity with the First Node:

- In the terminal where you started the second node (bob), verify that you see the following message indicating connectivity with the first node:

```
     2022-08-16 15:32:33 discovered:
12D3KooWEyoppNCUx8Yx66oV9fJnriXwCcXwDDUA2kj6vnc6iDEp
/ip4/127.0.0.1/tcp/30333
```

- This message confirms that the second node (bob) has discovered the first node (alice) and is connected to it.

5. View Blocks and Finalization Events:

- In the terminal where you started the first node (alice), you can view the blocks and finalization events by typing the following command:

```
     curl http://localhost:9933/ -H
"Content-Type:application/json;charset=utf-8" -d
```

```
'{"jsonrpc":"2.0","id":1,"method":"chain_getBlock"}'
```

- This command will display information about the latest block and its finalization status.
- You can repeat this command to observe the changes in block and finalization information as new blocks are produced and finalized.

Congratulations! You have successfully verified that blocks are being produced and finalized in the blockchain network consisting of two nodes.

Day 3

Add Trusted Nodes and authorise Specific Nodes

Adding Trusted Nodes

This module illustrates how to add trusted nodes to a blockchain network by creating a small, standalone network with a set of private validators.

As you learned in the Blockchain basics module, consensus is essential for all blockchains. Consensus refers to the agreement among network nodes on the state of data at a specific point in time. In this tutorial, we will use the Substrate node template, which employs a proof of authority consensus model known as the Aura consensus. This consensus protocol limits block production to a rotating list of authorized accounts, also known as authorities, who are considered trusted participants in the network.

The proof of authority consensus model offers a straightforward approach to starting a solo blockchain with a limited number of participants. In this module, you will learn how to generate the necessary keys to authorize a node's participation in the network, how to configure and share information about the network with other authorized accounts, and how to launch the network with an approved set of validators.

Module Objectives:

By completing this module, you will achieve the following objectives:

1. Generate key pairs for network authorities.
2. Create a custom chain specification file.
3. Launch a private two-node blockchain network.

Generate Your Account and Keys:

In the "Simulate a Network" module, you started peer nodes using predefined accounts and keys. However, for this module, we will generate our own secret keys for the validator nodes in the network. It is important to note that each participant in the blockchain network is responsible for generating unique keys.

There are several ways to generate key pairs, including using the node-template subcommand, the standalone subkey command-line program, the Polkadot-JS

application, or third-party key generation utilities. In this tutorial, we will generate keys using the Substrate node template and the key subcommand, although it is recommended to use an air-gapped computer that has never been connected to the internet when generating keys for a production blockchain. For simplicity, we will generate keys locally while remaining connected to the internet.

To generate keys using the node template:

1. Open a terminal shell on your computer.
2. Change to the root directory where you compiled the Substrate node template.
3. Generate a random secret phrase and keys by running the following command:

```
./target/release/node-template key generate --scheme Sr25519
--password-interactive
```

4. Enter a password for the generated keys when prompted.
5. The command will generate keys and display output similar to the following:

```
Secret phrase:  pig giraffe ceiling enter weird liar orange
decline behind total despair fly
Secret seed:
0x0087016ebbdcf03d1b7b2ad9a958e14a43f2351cd42f2f0a973771b90fb0112f
Public key (hex):
0x1a4cc824f6585859851f818e71ac63cf6fdc81018189809814677b2a4699cf45
Account ID:
0x1a4cc824f6585859851f818e71ac63cf6fdc81018189809814677b2a4699cf45
Public key (SS58):
5CfBuoHDvZ4fd8jkLQicNL8tgjnK8pVG9AiuJrsNrRAx6CNW
SS58 Address:
5CfBuoHDvZ4fd8jkLQicNL8tgjnK8pVG9AiuJrsNrRAx6CNW
```

Second Set Of Keys

To generate a second set of keys, you can follow the steps outlined below:

1. Open your preferred Substrate development environment or terminal.

2. Use the `subkey` command-line tool to generate the second set of keys. Run the following commands to generate the keys:

```
   subkey generate
```

This command generates a new key pair consisting of a secret seed and a corresponding public key.

3. Once the command is executed, you will receive an output similar to the following:

```
Secret phrase `brain torch diet secret myth tomato prison sort
invite order runway camp` is account:
   Secret seed:
0x82a6b3c9b5b7b23f812d057d95e1d6799b93d15e4de4a4d64b4f968ac5b69427
   Public key (hex):
0x0346d5fb0b04e2d60a6e0ae4ef201e97ed7e82805745faff455b4aafbc73826b
1c
```

Note: The above output is just an example. Your actual output will be different.

4. In the generated output, the secret seed is the private key, and the public key is derived from it. Make sure to keep the secret seed securely as it is required to sign transactions and authenticate the associated account.

5. Repeat the steps to generate a second set of keys using the same or different identity on your local computer, depending on your chosen method.

For your reference, the second set of keys used in this module are:

```
- Sr25519:
  - Secret seed:
0x82a6b3c9b5b7b23f812d057d95e1d6799b93d15e4de4a4d64b4f968ac5b69427
  - Public key:
0x0346d5fb0b04e2d60a6e0ae4ef201e97ed7e82805745faff455b4aafbc73826b
1c

- Ed25519:
  - Secret seed:
0x6b26eefaf737f123d3653db68d25c11ac3a0848b7f7f92955a21c5b4b3ce02f9
  - Public key:
0x52fd16c651c35d4e5f0e5772e8d2e6b23643dbcf6219b5db85046b0e01e1c9d8
```

Remember to use the generated keys accordingly in your Substrate network configuration or application.

Custom Chain Specification

To create a custom chain specification based on the local chain specification, you can follow the steps below:

1. Open a terminal shell on your computer.

2. Change to the root directory where you compiled the Substrate node template.

3. Export the local chain specification to a file named `customSpec.json` by running the following command:

```
   ./target/release/node-template build-spec
--disable-default-bootnode --chain local > customSpec.json
```

This command generates a `customSpec.json` file based on the local chain specification.

4. Open the `customSpec.json` file in a text editor.

5. Modify the `name` field to identify the chain specification as a custom testnet. For example:

```
   "name": "My Custom Testnet",
```

6. Modify the `aura` field to specify the nodes with the authority to create blocks. Add the Sr25519 SS58 address keys for each network participant. For example:

```
  "aura": {
     "authorities": [
       "5CfBuoHDvZ4fd8jkLQicNL8tgjnK8pVG9AiuJrsNrRAx6CNW",
       "5CXGP4oPXC1Je3zf5wEDkYeAqGcGXyKWSRX2Jm14GdME5Xc5"
     ]
   },
```

7. Modify the `grandpa` field to specify the nodes with the authority to finalize blocks. Add the Ed25519 SS58 address keys for each network participant. For example:

```
"grandpa": {
  "authorities": [
    [
      "5CuqCGfwqhjGzSqz5mnq36tMe651mU9Ji8xQ4JRuUTvPcjVN",
      1
    ],
    [
      "5DpdMN4bVTMy67TfMMtinQTcUmLhZBWoWarHvEYPM4jYziqm",
      1
    ]
  ]
},
```

Note: The second value in the `authorities` array represents the weight of the validator's vote. In this example, each validator has a weight of 1 vote.

8. Save your changes and close the file.

You have now created a custom chain specification based on the local chain specification. Make sure to share the `customSpec.json` file with the trusted network participants (validators) so they can use it to join your custom blockchain network.

Add Validators to the Custom Chain

To add validators to your custom chain specification, you can follow these steps:

1. Open the `customSpec.json` file that you modified earlier in a text editor.

2. Locate the `aura` section in the file. This section specifies the nodes with the authority to create blocks.

3. Add the Sr25519 SS58 address keys for each additional validator to the `authorities` array in the `aura` section. Make sure to use unique keys for each validator. For example:

```
  "aura": {
    "authorities": [
      "5CfBuoHDvZ4fd8jkLQicNL8tgjnK8pVG9AiuJrsNrRAx6CNW",
      "5CXGP4oPXC1Je3zf5wEDkYeAqGcGXyKWSRX2Jm14GdME5Xc5",
      "Additional_Sr25519_Address_1",
      "Additional_Sr25519_Address_2",
      ...
    ]
  },
```

4. Locate the `grandpa` section in the file. This section specifies the nodes with the authority to finalize blocks.

5. Add the Ed25519 SS58 address keys for each additional validator to the `authorities` array in the `grandpa` section. Additionally, include a voting weight for each validator. For example:

```
"grandpa": {
    "authorities": [
      [
        "5CuqCGfwqhjGzSqz5mnq36tMe651mU9Ji8xQ4JRuUTvPcjVN",
        1
      ],
      [
        "5DpdMN4bVTMy67TfMMtinQTcUmLhZBWoWarHvEYPM4jYziqm",
        1
      ],
      [
        "Additional_Ed25519_Address_1",
        1
      ],
      [
        "Additional_Ed25519_Address_2",
        1
      ],
      ...
    ]
  },
```

Note: Make sure to assign a unique voting weight to each validator. In the example above, each validator has a weight of 1 vote.

6. Save your changes and close the file.

By adding the additional validators' addresses and specifying their authority roles and voting weights in the `aura` and `grandpa` sections, you have successfully added validators to your custom chain specification. Remember to share the updated `customSpec.json` file with the validators so they can join your blockchain network.

Convert to Chain Specification and Prepare to launch Private Net

To convert the chain specification to the raw format and prepare to launch the private network, you can follow these steps:

1. Open a terminal shell on your computer.

2. Change to the root directory where you compiled the Substrate node template.

3. Convert the `customSpec.json` chain specification to the raw format with the file name `customSpecRaw.json` by running the following command:

```
   ./target/release/node-template build-spec
--chain=customSpec.json --raw --disable-default-bootnode >
customSpecRaw.json
```

This command converts the chain specification to the raw format and saves it in the `customSpecRaw.json` file.

4. Distribute the `customSpecRaw.json` file to all the network participants. This ensures that each node stores the data using the proper storage keys.

Now that you have converted the chain specification to the raw format and distributed it to the network participants, you are ready to launch your private blockchain.

Ensure that you have completed the following tasks:

- Generated or collected the account keys for at least two authority accounts.
- Updated your custom chain specification to include the keys for block production (aura) and block finalization (grandpa).
- Convert your custom chain specification to the raw format and distribute the raw chain specification to the nodes participating in the private network.

Start your First Node

To start the first node in your private blockchain network, follow these steps:

1. Open a terminal shell on your computer.

2. Change to the root directory where you compiled the Substrate node template.

3. Start the first node using the custom chain specification by running a command similar to the following:

```
./target/release/node-template \
  --base-path /tmp/node01 \
  --chain ./customSpecRaw.json \
  --port 30333 \
  --ws-port 9945 \
  --rpc-port 9933 \
  --telemetry-url "wss://telemetry.polkadot.io/submit/0" \
  --validator \
  --rpc-methods Unsafe \
  --name MyNode01 \
  --password-interactive
```

In this command, you specify various options:

- `--base-path /tmp/node01`: Specifies the custom location for the chain associated with this first node.
- `--chain ./customSpecRaw.json`: Specifies the custom chain specification.
- `--port 30333`: Specifies the port on which the node will listen for incoming connections.
- `--ws-port 9945`: Specifies the WebSocket port for the node.
- `--rpc-port 9933`: Specifies the RPC port for the node.
- `--telemetry-url "wss://telemetry.polkadot.io/submit/0"`: Specifies the telemetry URL for monitoring the node's performance.
- `--validator`: Indicates that this node is an authority for the chain.
- `--rpc-methods Unsafe`: Allows you to continue the tutorial using an unsafe communication mode (not recommended for production).
- `--name MyNode01`: Gives your node a human-readable name in the telemetry UI.
- `--password-interactive`: Prompts you to enter the keystore password you used to generate the node01 keys.

Make sure to replace `/tmp/node01` with the desired base path for your node and adjust the port numbers as needed.

4. When prompted, enter the keystore password you used to generate the node01 keys.

This password is required to access the keystore, which contains the node's keys.

5. Take note of the information displayed in the terminal shell. It should include details such as the chain specification, node name, node identity, and network connectivity status.

For example:

```
📋 Chain specification: My Custom Testnet
🏷️ Node name: MyNode01
👤 Role: AUTHORITY
...
```

This information will be needed when connecting other nodes to the network.

6. Once you have taken note of the necessary information, you can stop the node by pressing `Control-c` in the terminal shell.

Congratulations! You have successfully started the first node in your private blockchain network. Next, you'll need to add your keys to the keystore so that they can be used by the node for block production and finalization.

Add Keys to Keystore

To add keys to the keystore for each node, follow these steps:

For the first node:

1. Open a terminal shell on your computer.
2. Change to the root directory where you compiled the Substrate node template.
3. Insert the aura secret key by running the following command:

```
./target/release/node-template key insert --base-path /tmp/node01
\
  --chain customSpecRaw.json \
  --scheme Sr25519 \
  --suri "pig giraffe ceiling enter weird liar orange decline
behind total despair fly" \
  --password-interactive \
  --key-type aura
```

4. Type the password you used to generate the keys.

5. Insert the grandpa secret key by running the following command:

./target/release/node-template key insert \

--base-path /tmp/node01 \

--chain customSpecRaw.json \

--scheme Ed25519 \

--suri "pig giraffe ceiling enter weird liar orange decline behind total despair fly" \

--password-interactive \

--key-type gran

6. Type the password you used to generate the keys.

7. Verify that your keys are in the keystore for node01 by running the following command

```
ls /tmp/node01/chains/local_testnet/keystore
```

For the second node:

1. Open a terminal shell on the second computer.

2. Change to the root directory where you compiled the Substrate node template.

3. Start the second blockchain node by running the following command:

```
./target/release/node-template \
  --base-path /tmp/node02 \
  --chain ./customSpecRaw.json \
  --port 30334 \
  --ws-port 9946 \
  --rpc-port 9934 \
  --telemetry-url "wss://telemetry.polkadot.io/submit/ 0" \
  --validator \
  --rpc-methods Unsafe \
  --name MyNode02 \
  --bootnodes
/ip4/127.0.0.1/tcp/30333/p2p/12D3KooWLmrYDLoNTyTYtRdDyZLWDe1paxzxT
w5RgjmHLfzW96SX \
  --password-interactive
```

4. Insert the aura secret key by running the following command:

```
./target/release/node-template key insert --base-path /tmp/node02
\
  --chain customSpecRaw.json \
  --scheme Sr25519 \
  --suri "<second-participant-secret-seed>" \
  --password-interactive \
  --key-type aura
```

Replace `<second-participant-secret-seed>` with the secret phrase or secret seed for the second key pair you generated.

5. Type the password you used to generate the keys.

6. Insert the grandpa secret key by running the following command:

```
./target/release/node-template key insert --base-path /tmp/node02
\
  --chain customSpecRaw.json \
  --scheme Ed25519 \
  --suri "<second-participant-secret-seed>" \
  --password-interactive \
  --key-type gran
```

Replace `<second-participant-secret-seed>` with the secret phrase or secret seed for the second key pair you generated.

7. Type the password you used to generate the keys.

8. Verify that your keys are in the keystore for node02 by running the following command:

```
ls /tmp/node02/chains/local_testnet/keystore
```

After adding the keys to the keystore for both nodes, you can restart the nodes by using the commands mentioned in the previous steps. Once the nodes are restarted, you should see blocks being produced and finalized on both nodes.

Authorize specific nodes

In Add trusted nodes, you saw how to build a simple network with a known set of validator nodes. That tutorial illustrated a simplified version of a **permissioned network**. In a permissioned network, only **authorized nodes** are allowed to perform specific network activities. For example, you might grant some nodes the permission to validate blocks and other nodes the permission to propagate transactions.

A blockchain with nodes that are granted specific permissions is different from a **public** or **permissionless** blockchain. In a permissionless blockchain, anyone can join the network by running the node software on suitable hardware. In general, a permissionless blockchain offers greater decentralization of the network. However, there are use cases where creating a permissioned blockchain might be appropriate. For example, a permissioned blockchain would be suitable for the following types of projects:

- For a private or consortium network, such as a private enterprise or a non-profit organization.
- In highly-regulated data environments, such as healthcare, finance, or business-to-business ledgers.
- For testing of a pre-public blockchain network at scale.

This tutorial illustrates how you can build a permissioned network with Substrate by using the node authorization pallet.

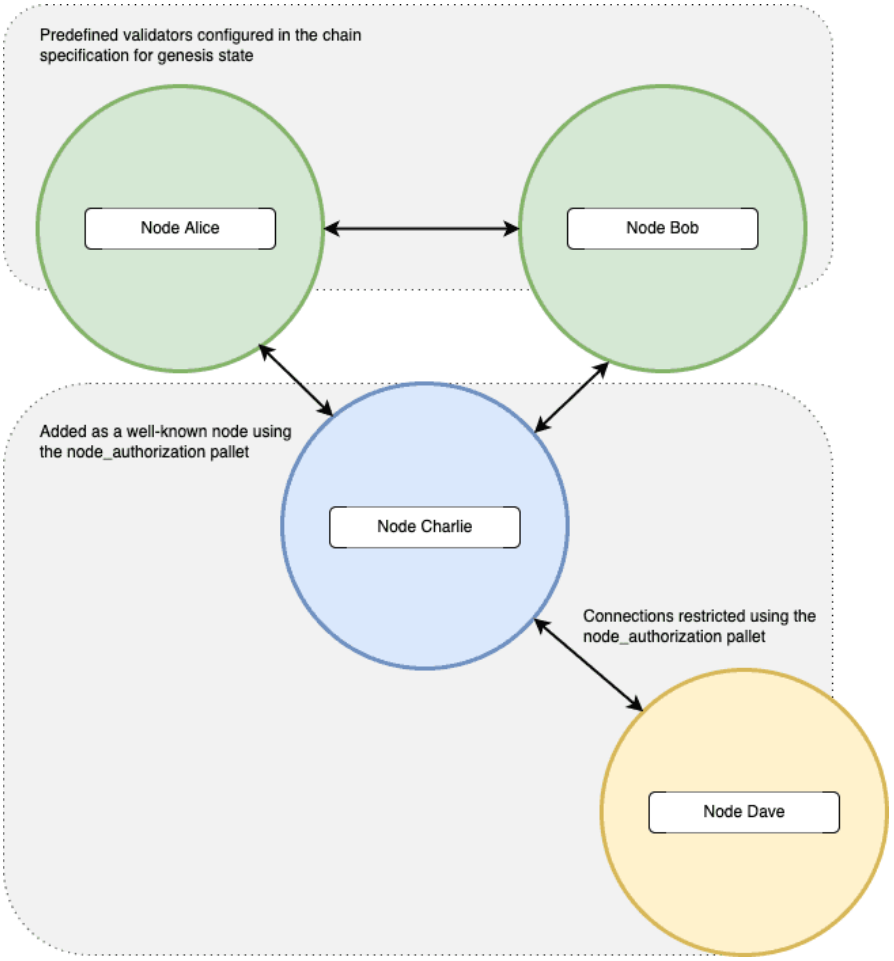
Node authorization and ownership

The `node-authorization` pallet is a prebuilt FRAME pallet that enables you to manage a configurable set of nodes for a network. Each node is identified by a peer identifier (`PeerId`). Each peer identifier is owned by **one and only one** account owner (`AccountId`) that claims the node.

There are two ways you can authorize a node to join the network:

- By adding the peer identifier to the list of predefined nodes in the chain specification file as part of the genesis configuration for the chain. You must be approved by the governance mechanism for the chain or have access to the root account for the Sudo pallet in the network to do this.
- By asking for a *paired peer* connection from a specific node. You can add connections between nodes by using predefined node peer identifiers or by using the peer identifiers generated from the public and private keys for each of the nodes.

As the following diagram suggests, this tutorial illustrates both authorization methods, with the peer identifiers for Alice and Bob defined in the chain specification and the additional nodes added using the node authorization pallet.



Note that *any* user can claim to be the owner of a `PeerId`. To protect against false claims, you should claim the node identifier *before* you start the node. After you start the node, its `PeerID` is visible to the network and *anyone* could subsequently claim it.

As the owner of a node, you can add and remove connections for your node. For example, you can manipulate the connection between a predefined node and your node or between your node and other non-predefined nodes. You can't change the connections for predefined nodes. They are always allowed to connect with each other.

The `node-authorization` pallet uses an offchain worker to configure its node connections. Make sure to enable the offchain worker when you start the node because it is disabled by default for non-authority nodes.

Before you begin

Before you begin, verify the following:

- You have configured your environment for Substrate development by installing Rust and the Rust toolchain.
- You have completed Build a local blockchain and have the Substrate node template installed locally.
- You have completed the Add trusted nodes tutorial.
- You are generally familiar with peer-to-peer networking in Substrate.

Module objectives

By completing this module, you will accomplish the following objectives:

- Check out and compile the node template.
- Add the node authorization pallet to the node template runtime.
- Launch multiple nodes and authorize new nodes to join.

Build the node template

If you have completed previous tutorials, you should have the Substrate node template repository available locally.

1. Open a terminal shell on the computer where you have Substrate node template repository.
2. Change to the root of the node template directory, if necessary, by running the following command:
   `cd substrate-node-template`
3. Switch to a working branch for the repository if you want to save your changes by running a command similar to the following:

```
git switch -c my-wip-branch
```

4. Compile the node template by running the following command:

```
cargo build --release
```

The node template should compile without any errors. If you encounter issues when you compile, you can try the troubleshooting tips in Troubleshoot Rust issues.

Add the node authorization pallet

Before you can use a new pallet, you must add some information about it to the configuration file that the compiler uses to build the runtime binary.

For Rust programs, you use the `Cargo.toml` file to define the configuration settings and dependencies that determine what gets compiled in the resulting binary. Because the Substrate runtime compiles to both a native Rust binary that includes standard library functions and a WebAssembly (Wasm) binary that does not include the standard library, the `Cargo.toml` file controls two important pieces of information:

- The pallets to be imported as dependencies for the runtime, including the location and version of the pallets to import.
- The features in each pallet that should be enabled when compiling the native Rust binary. By enabling the standard (`std`) feature set from each pallet, you can compile the runtime to include functions, types, and primitives that would otherwise be missing when you build the WebAssembly binary.

For general information about adding dependencies in `Cargo.toml` files, see Dependencies in the Cargo documentation. For information about enabling and managing features from dependent packages, see Features in the Cargo documentation.

Add node-authorization dependencies

To add the `node-authorization` pallet to the Substrate runtime:

1. Open a terminal shell and change to the root directory for the node template.
2. Open the `runtime/Cargo.toml` configuration file in a text editor.

Locate the `[dependencies]` section and add the `pallet-node-authorization` crate to make it available to the node template runtime.

[dependencies]

```
pallet-node-authorization = { default-features = false, version =
"4.0.0-dev", git = "https://github.com/paritytech/substrate.git",
branch = "polkadot-v0.9.28" }
```

This line imports the `pallet-node-authorization` crate as a dependency and specifies the following configuration details for the crate:

- The pallet features are not enabled by default when compiling the runtime.
- The version identifier for the crate.
- The repository location for retrieving the `pallet-node-authorization` crate.

- The branch for retrieving the crate.

3. Note that you should use the same branch and version information for all pallets to ensure that they are compatible with each other. Using pallets from different branches can result in compiler errors. This example illustrates adding pallets to the `Cargo.toml` file if the other pallets use `branch = "polkadot-v0.9.28"`.

Add the `pallet-node-authorization/std` features to the list of `features` to enable when compiling the runtime.

```
[features]
default = ['std']
std = [
 ...
 "pallet-node-authorization/std",    # add this line
 ...
]
```

This section specifies the default feature set to compile for this runtime is the `std` features set. When the runtime is compiled using the `std` feature set, the `std` features from all of the pallets listed as dependencies are enabled. For more detailed information about how the runtime is compiled as a native Rust binary with the standard library and as a WebAssembly binary using the `no_std` attribute, see Build process.

If you forget to update the `features` section in the `Cargo.toml` file, you might see `cannot find function` errors when you compile the runtime binary

.

4. Check that the new dependencies resolve correctly by running the following command:

```
cargo check -p node-template-runtime --release
```

Add an administrative rule

To simulate governance in this tutorial, you can configure the pallet to use the `EnsureRoot` privileged function that can be called using the Sudo pallet. The Sudo pallet is included in the node template by default and enables you to make calls through the root-level administrative account. In a production environment, you would use more realistic governance-based checking.

To enable the `EnsureRoot` rule in your runtime:

1. Open the `runtime/src/lib.rs` file in a text editor.
2. Add the following line to the file:

```
use frame_system::EnsureRoot;
```

Implement the Config trait

Every pallet has a Rust trait called `Config`. The `Config` trait is used to identify the parameters and types that the pallet needs.

Most of the pallet-specific code required to add a pallet is implemented using the `Config` trait. You can review what you need to implement for any pallet by referring to its Rust documentation or the source code for the pallet. For example, to see what you need to implement for the `Config` trait in the node-authorization pallet, you can refer to the Rust documentation for `pallet_node_authorization::Config`.

To implement the `node-authorization` pallet in your runtime:

Open the `runtime/src/lib.rs` file in a text editor.

Add the `parameter_types` section for the pallet using the following code:

```
 parameter_types! {
 pub const MaxWellKnownNodes: u32 = 8;
 pub const MaxPeerIdLength: u32 = 128;
}
```

Add the `impl` section for the `Config` trait for the pallet using the following code:

```
 impl pallet_node_authorization::Config for Runtime {
 type RuntimeEvent = RuntimeEvent;
 type MaxWellKnownNodes = MaxWellKnownNodes;
 type MaxPeerIdLength = MaxPeerIdLength;
 type AddOrigin = EnsureRoot<AccountId>;
 type RemoveOrigin = EnsureRoot<AccountId>;
 type SwapOrigin = EnsureRoot<AccountId>;
 type ResetOrigin = EnsureRoot<AccountId>;
 type WeightInfo = ();
}
```

Add the pallet to the `construct_runtime` macro with the following line of code:

```
construct_runtime!(
pub enum Runtime where
   Block = Block,
   NodeBlock = opaque::Block,
   UncheckedExtrinsic = UncheckedExtrinsic
 {
   /*** Add This Line ***/
   NodeAuthorization: pallet_node_authorization::{Pallet, Call,
Storage, Event<T>, Config<T>},
 }
```

```
);
```

Save your changes and close the file.

Check that the configuration can be compiled by running the following command:\

```
cargo check -p node-template-runtime --release
```

Add genesis storage for authorized nodes

Before you can launch the network to use node authorization, some additional configuration is needed to handle the peer identifiers and account identifiers. For example, the `PeerId` is encoded in bs58 format, so you need to add a new dependency for the bs58 library in the `node/Cargo.toml` to decode the `PeerId` to get its bytes. To keep things simple, the authorized nodes are associated with predefined accounts.

To configure genesis storage for authorized nodes:

Open the `node/Cargo.toml` file in a text editor.

Locate the `[dependencies]` section and add the `bs58` library to the node template.

```
[dependencies]
bs58 = { version = "0.4.0" }
```

Save your changes and close the file.

Open the `node/src/chain_spec.rs` file in a text editor.

Add genesis storage for nodes that are authorized to join the network using the following code:

use sp_core::OpaquePeerId; // A struct wraps Vec<u8> to represent the node `PeerId`.

use node_template_runtime::NodeAuthorizationConfig; // The genesis config that serves the pallet.

Locate the `testnet_genesis` function that configures initial storage state for FRAME modules.

For example:

```
 /// Configure initial storage state for FRAME modules.
fn testnet_genesis(
 wasm_binary: &[u8],
 initial_authorities: Vec<(AuraId, GrandpaId)>,
 root_key: AccountId,
 endowed_accounts: Vec<AccountId>,
 _enable_println: bool,
 ) -> GenesisConfig {
//Within the GenesisConfig declaration, add the following code block:
 node_authorization: NodeAuthorizationConfig {
   nodes: vec![
     (

OpaquePeerId(bs58::decode("12D3KooWBmAwcd4PJNJvfV89HwE48nwkRmAgo8Vy3uQEyNNHBox2").into_vec().unwrap()),
       endowed_accounts[0].clone()
     ),
     (

OpaquePeerId(bs58::decode("12D3KooWQYV9dGMFoRzNStwpXztXaBUjtPqi6aU76ZgUriHhKust").into_vec().unwrap()),
       endowed_accounts[1].clone()
     ),
   ],
 },
```

In this code, `NodeAuthorizationConfig` contains a `nodes` property, which is a vector with a tuple of two elements. The first element of the tuple is the `OpaquePeerId`. The `bs58::decode` operation converts the human-readable `PeerId`—for example, `12D3KooWBmAwcd4PJNJvfV89HwE48nwkRmAgo8Vy3uQEyNNHBox2`—to bytes. The second element of the tuple is the `AccountId` that represents the owner of this node. This example uses the predefined Alice and Bob, identified here as endowed accounts [0] and [1].

Save your changes and close the file.

Verify that the node compiles

Now that you have completed the code changes, you are ready to verify that the node compiles.

To compile the node:

1. Change to the root of the `substrate-node-template` directory, if necessary:
2. Compile the node by running the following command:

```
cargo build --release
```

If there are no syntax errors, you are ready to proceed. If there are errors, follow the instructions in the compile output to fix them then rerun the `cargo build` command.

Identify account keys to use

You have already configured the nodes associated with the Alice and Bob accounts in genesis storage. You can use the `subkey` program to inspect the keys associated with predefined accounts and to generate and inspect your own keys. However, if you run the `subkey generate-node-key` command, your node key and peer identifier are randomly generated and won't match the keys used in the tutorial. Because this

tutorial uses predefined accounts and well-known node keys, you can use the following keys for each account.

Alice

Key type	Key value
Node key	c12b6d18942f5ee8528c8e2baf4e147b5c5c18710926ea492d09cbd9f6c9f82a
Peer identifier generated from the node key	12D3KooWBmAwcd4PJNJvfV89HwE48nwkRmAgo8Vy3uQEyNNHBox2
Peer identifier decoded to hex	0x0024080112201ce5f00ef6e89374afb625f1ae4c1546d31234e87e3c3f51a62b91dd6bfa57df

Bob

Key type	Key value
Node key	6ce3be907dbcabf20a9a5a60a712b4256a54196000a8ed4050d352bc113f8c58
Peer identifier generated from the node key	12D3KooWQYV9dGMFoRzNStwpXztXaBUjtPqi6aU76ZgUriHhKust
Peer identifier decoded to hex	0x002408011220dacde7714d8551f674b8bb4b54239383c76a2b286fa436e93b2b7eb226bf4de7

The two other development accounts—Charlie and Dave—don't have well-known node keys or peer identifiers defined in the genesis configuration. For demonstration purposes, you can use the following keys for these accounts.

Charlie

Key type	Key value
Node key	3a9d5b35b9fb4c42aafadeca046f6bf56107bd2579687f069b42646684b94d9e
Peer identifier generated from the node key	12D3KooWJvyP3VJYymTqG7eH4PM5rN4T2agk5cdNCfNymAqwqcvZ
Peer identifier decoded to hex	0x002408011220876a7b4984f98006dc8d666e28b60de307309835d775e7755cc770328cdacf2e

Dave

Key type	Key value
Node key	a99331ff4f0e0a0434a6263da0a5823ea3afcfffe590c9f3014e6cf620f2b19a
Peer identifier generated from the node key	12D3KooWPHWFrfaJzxPnqnAYAoRUyAHHKqACmEycGTVmeVhQYuZN
Peer identifier decoded to hex	0x002408011220c81bc1d7057a1511eb9496f056f6f53cdfe0e14c8bd5ffca47c70a8d76c1326d

For this tutorial, you can copy the node key to a file, then use the `subkey inspect-node-key` to verify the peer identifiers for Charlie and Dave. For example, save the node key for Charlie to a file named `charlie-node-key` by using a command like the following:

```
echo -n
"3a9d5b35b9fb4c42aafadeca046f6bf56107bd2579687f069b42646684b94d9e"
> charlie-node-key
```

You can then run the following command to verify the peer identifier:

```
./subkey inspect-node-key --file charlie-node-key
```

The command displays the peer identifier for the Charlie node:

```
12D3KooWJvyP3VJYymTqG7eH4PM5rN4T2agk5cdNCfNymAqwqcvZ
```

If you generate node keys for your own account, save the peer identifier for the node to a file so you can pass it to `subkey inspect-node-key` or other commands when needed.

Launch network nodes

You can now use the node keys and peer identifiers for the predefined accounts to launch the permissioned network and authorize other nodes to join.

For the purposes of this tutorial, you are going to launch four nodes. Three of the nodes are associated with predefined accounts and all three of those nodes are allowed to author and validate blocks. The fourth node is a **sub-node** that is only authorized to read data from a selected node with the approval of that node's owner.

Start the first node

Because you have configured genesis storage to use the well-known node keys for Alice and Bob, you can use the `--alice` command shortcut for `--name alice --validator` to start the first node.

To start the first node:

1. Open a terminal shell on your computer, if necessary.
2. Change to the root directory where you compiled the Substrate node template.Start the first node by running the following command:

```
 ./target/release/node-template \
--chain=local \
--base-path /tmp/validator1 \
--alice \
--node-key=c12b6d18942f5ee8528c8e2baf4e147b5c5c18710926ea492d09cbd
9f6c9f82a \
--port 30333 \
```

```
--ws-port 9944
```

3. In this command, the `--node-key` option to specify the key to be used for a secure connection to the network. This key is also used internally to generate the human-readable PeerId as shown in above section
   As you might have seen in other modules, the command-line options used are:
   - `--chain=local` for a local testnet (not the same as the `--dev` flag!).
   - `--alice` to name the node `alice` and make the node a validator that can author and finalize blocks.
   - `--port` to assign a port for peer-to-peer communication.
   - `--ws-port` to assign a listening port for WebSocket connections.

Start the second node

You can start the second node using the `--bob` command shortcut for `--name bob --validator` to start the second node.

To start the second node:

1. Open a **new** terminal shell on your computer.
2. Change to the root directory where you compiled the Substrate node template.

the second node by running the following command:

```
 ./target/release/node-template \
--chain=local \
--base-path /tmp/validator2 \
--bob \
--node-key=6ce3be907dbcabf20a9a5a60a712b4256a54196000a8ed4050d352b
c113f8c58 \
--bootnodes
/ip4/127.0.0.1/tcp/30333/p2p/12D3KooWBmAwcd4PJNJvfV89HwE48nwkRmAgo
8Vy3uQEyNNHBox2 \
--port 30334 \
```

```
-ws-port 9945
```

After both nodes are started, you should start to see new blocks authored and finalized in both terminal logs.

Add a third node to the list of well-known nodes

You can start the third node with the `--name charlie` command. The `node-authorization` pallet uses an offchain worker to configure node connections. Because the third node is not a well-known node and it will have the fourth node in the network configured as a read-only sub-node, you must include the command-line option to enable the offchain worker.

To start the third node:

1. Open a **new** terminal shell on your computer.
2. Change to the root directory where you compiled the Substrate node template.

Start the third node by running the following command:

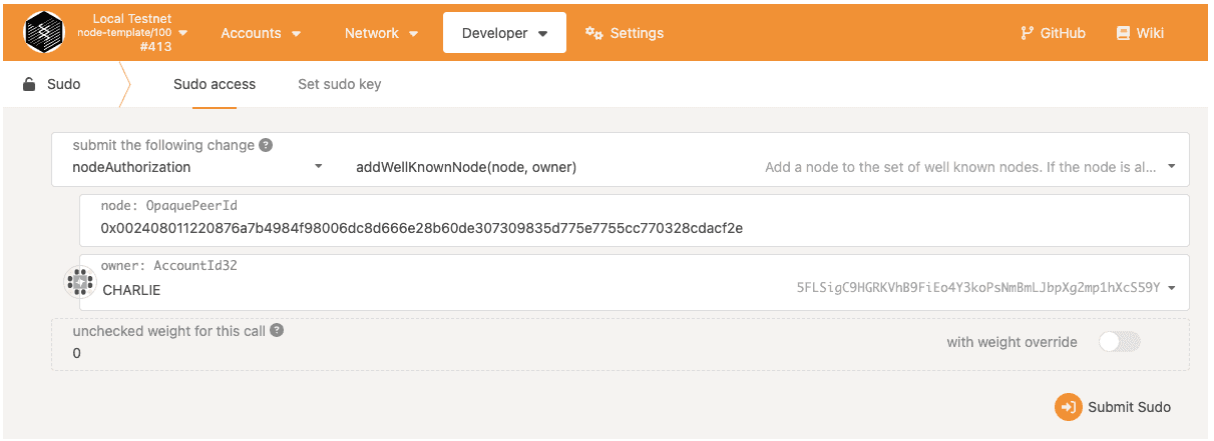
```
./target/release/node-template \
--chain=local \
--base-path /tmp/validator3 \
--name charlie  \
--node-key=3a9d5b35b9fb4c42aafadeca046f6bf56107bd2579687f069b42646
684b94d9e \
--port 30335 \
--ws-port=9946 \
--offchain-worker always
```

After you start this node, you should see there are **no connected peers** for the node. Because this is a permissioned network, the node must be explicitly authorized to connect. The Alice and Bob nodes were configured in the genesis `chain_spec.rs` file. All other nodes must be added manually using a call to the Sudo pallet.

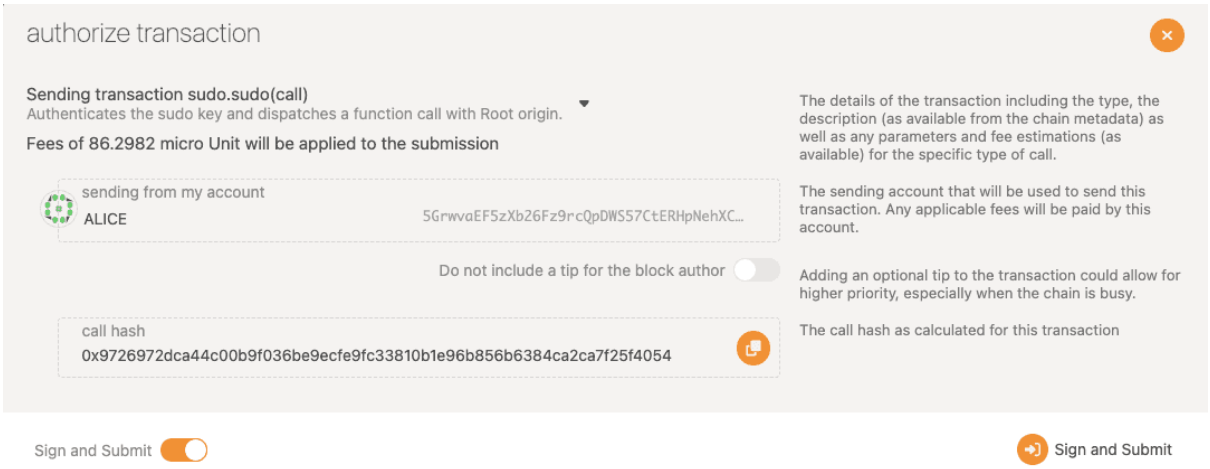
Authorize access for the third node

This tutorial uses the Sudo pallet for governance. Therefore, you can use the Sudo pallet to call the `addWellKnownNode` function provided by `node-authorization` pallet to add the third node. To keep things simple, you can use the Polkadot/Substrate Portal application to access the Sudo pallet.

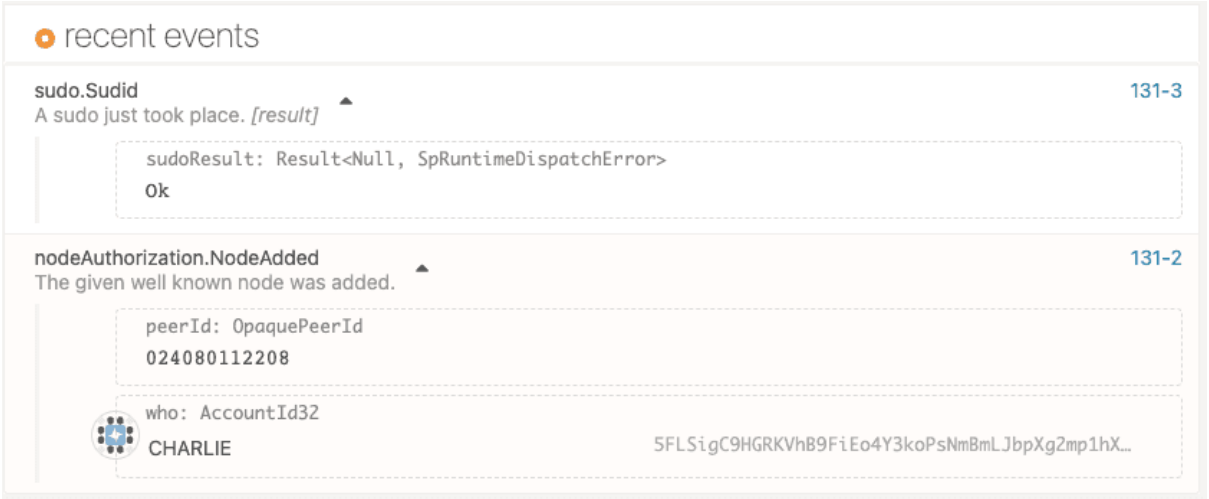
1. Open the Polkadot/Substrate Portal in a browser.
2. Click **Developer** and select **Sudo**.
3. Select **nodeAuthorization** and select **addWellKnownNode(node, owner)**.
4. Copy and paste the hex-encoded peer identifier for the node owned by Charlie after the required 0x prefix.
5. Select **Charlie** as the node owner.
6. Click **Submit Sudo**.



7. In Authorize transaction, note that the Alice development account is the default root administrative account and used as the `sudo` origin for this call, then click **Sign and Submit**.



8. Click **Network** and select **Explorer** to view the recent transactions.



After the transaction is included in the block, you should see the `charlie` node is connected to the `alice` and `bob` nodes, and starts to sync blocks. The three nodes can find each other using the mDNS discovery mechanism that is enabled by default in a local network. Also ensure any local firewall is configured to allow mDNS.

If your nodes are not on the same local network, you should use the command-line option `--no-mdns` to disable it.

Allow connections from a sub-node

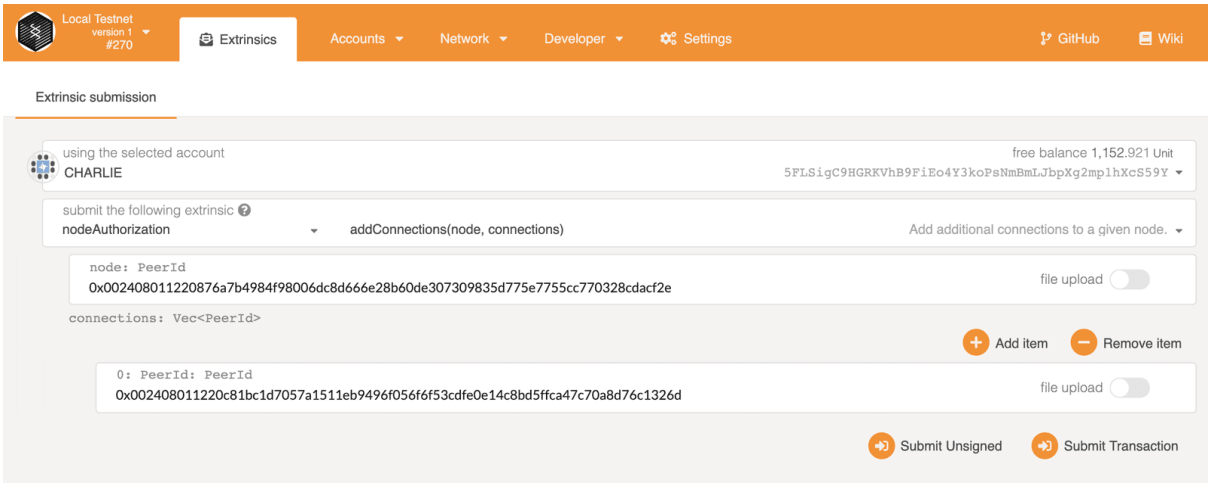
The fourth node in this network is not going to be used as a validator or added to the list of well-known nodes. This fourth node is owned by the `dave` user account, but is a sub-node of the `charlie` node. The sub-node can only access the network by connecting to the node owned by the `charlie` parent node. It's important to remember that the parent node is responsible for any sub-node it authorizes to connect to the network. The owner of the parent node is responsible for controlling access if the sub-node needs to be removed or audited.

Because this is a *permissioned network*, Charlie must configure his node to allow the connection from the node owned by Dave. You can use the Polkadot/Substrate Portal application to grant this permission.

To allow the sub-node to access the network:

1. Open the Polkadot/Substrate Portal in a browser.
2. Click **Developer** and select **Extrinsics**.
3. Select **nodeAuthorization** and select **addConnections(node, connections)**.
4. Copy and paste the hex-encoded peer identifier for the node owned by Charlie after the required 0x prefix.
5. For the connections parameter, copy and paste the hex-encoded peer identifier for the node owned by Dave after the required 0x prefix, then click **Submit Transaction**.

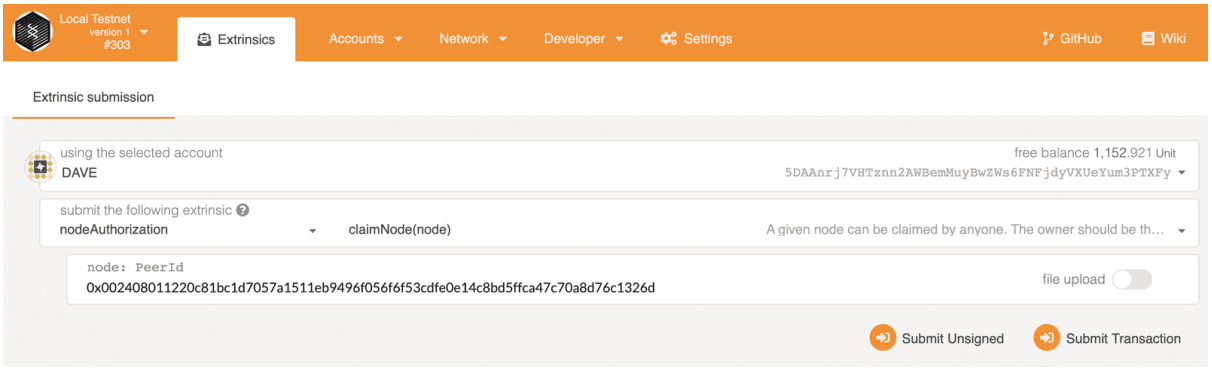
6. Review the transaction details, then click **Sign and Submit**.



Claim the sub-node

Before starting the sub-node, the node owner should claim the peer identifier for his node. You can use the Polkadot/Substrate Portal application to submit a transaction to claim the node owned by the `dave` account.

1. Open the Polkadot/Substrate Portal in a browser.
2. Click **Developer** and select **Extrinsics**.
3. Select **nodeAuthorization** and select **claimNode(node)**.
4. Copy and paste the hex-encoded peer identifier for the node owned by Dave after the required 0x prefix, then click **Submit Transaction**.
5. Review the transaction details, then click **Sign and Submit**.



Start the sub-node

After claiming the node peer identifier, you are ready to start the sub-node.

To start the sub-node:

1. Open a **new** terminal shell on your computer.
2. Change to the root directory where you compiled the Substrate node template.

Start the sub-node by running the following command:

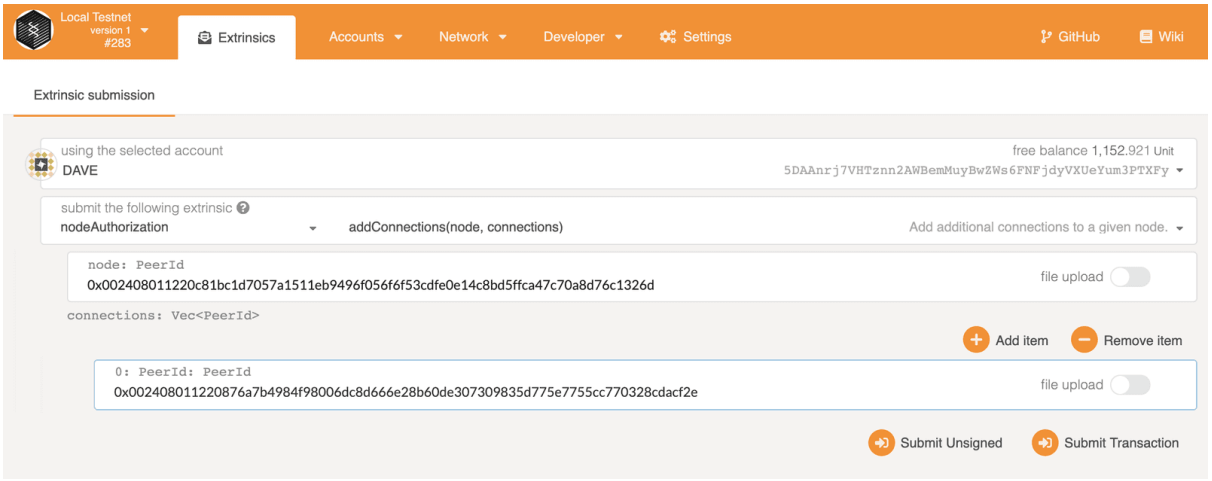
```
./target/release/node-template \
--chain=local \
--base-path /tmp/validator4 \
--name dave \
--node-key=a99331ff4f0e0a0434a6263da0a5823ea3afcfffe590c9f3014e6cf
620f2b19a \
--port 30336 \
--ws-port 9947 \
--offchain-worker always
```

Allow connections to the sub-node

You now have a network with four nodes. However, to allow the sub-node to participate, you must configure it to allow connections from the parent node owned by Charlie. The steps are similar to the ones you previously performed to allow connections from the node owned by Dave.

1. Open the Polkadot/Substrate Portal in a browser.
2. Click **Developer** and select **Extrinsics**.
3. Select **nodeAuthorization** and select **addConnections(node, connections)**.
4. Copy and paste the hex-encoded peer identifier for the node owned by Dave after the required 0x prefix.
5. For the connections parameter, copy and paste the hex-encoded peer identifier for the node owned by Charlie after the required 0x prefix, then click **Submit Transaction**.

6. Review the transaction details, then click **Sign and Submit**.



You should now see the sub-node has only one peer—the node that belongs to Charlie—and is synchronizing blocks from the chain. If the sub-node doesn't connect to its peer node right away, try stopping and restarting the sub-node.

Keys required to submit transactions

You should note that any account can be used to sign and submit transaction that affect the behavior of other nodes. However, to sign and submit a transaction that affects a node you don't own:

- The transaction must reference *on chain data*.
- You must have the *signing key* for an account with the required origin available in the keystore.

In this tutorial, all of the nodes have access to the development account signing keys. Therefore, you were able to sign and submit transactions that affected any connected node using account to act on behalf of Charlie or Dave. If you were building a permissioned network for a real world application, node operators would most likely only have access to their own node keys, and node owner accounts would be required to sign and submit transactions that affect the node where they have control of the signing key.

DAY-4

Upgrade a running network

Unlike many blockchains, the Substrate development framework supports **forkless upgrades** to the runtime that is the core of the blockchain. Most blockchain projects require a hard fork of the code base to support ongoing development of new features or enhancements to existing features. With Substrate, you can deploy enhanced runtime capabilities—including breaking changes—without a hard fork. Because the definition of the runtime is itself an element in the state of a Substrate-based chain, network participants can update this value by calling the `set_code` function in a transaction. Because updates to the runtime state are validated using the blockchain's consensus mechanisms and cryptographic guarantees, network participants can use the blockchain itself to distribute updated or extended runtime logic without needing to fork the chain or release a new blockchain client.

This module illustrates how to upgrade the runtime without creating a fork of the code base or stopping the progress of the chain. In this tutorial, you'll make the following changes to a Substrate runtime on a running network node:

- Increase the `spec_version` of the runtime.
- Add the Utility pallet to the runtime.
- Increase the minimum balance for network accounts.

Before you begin

- You have configured your environment for Substrate development by installing Rust and the Rust toolchain.
- You have completed Build a local blockchain and have the Substrate node template installed locally.
- You have reviewed Add a pallet to the runtime for an introduction to adding a new pallet to the runtime.

Module objectives

By completing this tutorial, you will accomplish the following objectives:

- Use the Sudo pallet to simulate governance for a chain upgrade.
- Upgrade the runtime for a running node to include a new pallet.
- Submit a transaction to upload the modified runtime onto a running node.

Authorize an upgrade with Sudo

Typically, runtime upgrades are managed through governance with community members voting to approve or reject upgrade proposals. In place of governance, this tutorial uses the Sudo pallet and the `Root` origin to identify the runtime administrator with permission to upgrade the runtime. Only this root-level administrator can update the runtime by calling the `set_code` function. The Sudo pallet enables you to invoke the `set_code` function using the `Root` origin by specifying the account that has root-level administrative permissions.

By default, the chain specification file for the node template specifies that the `alice` development account is the owner of the Sudo administrative account. Therefore, this tutorial uses the `alice` account to perform runtime upgrades.

Resource accounting for runtime upgrades

Function calls that are dispatched to the Substrate runtime are always associated with a weight to account for resource usage. The FRAME System module sets boundaries on the block length and block weight that these transactions can use. However, the `set_code` function is intentionally designed to consume the maximum weight that can fit in a block. Forcing a runtime upgrade to consume an entire block prevents transactions in the same block from executing on different versions of a runtime.

The weight annotation for the `set_code` function also specifies that the function is in the `Operational` class because it provides network capabilities. Function calls that are identified as operational:

- Can consume the entire weight limit of a block.
- Are given maximum priority.
- Are exempt from paying transaction fees.

Managing resource accounting

In this tutorial, the `sudo_unchecked_weight` function is used to invoke the `set_code` function for the runtime upgrade. The `sudo_unchecked_weight` function is the same as the `sudo` function except that it supports an additional parameter to specify the weight to use for the call. This parameter enables you to work around resource accounting safeguards to specify a weight of zero for the call that dispatches the `set_code` function. This setting allows for a block to take *an indefinite time to compute* to ensure that the runtime upgrade does not fail, no matter how complex the operation is. It can take all the time it needs to succeed or fail.

Add the Utility pallet to the runtime

By default, the node template doesn't include the Utility pallet in its runtime. To illustrate a runtime upgrade, you can add the Utility pallet to a running node.

Start the local node

This tutorial illustrates how to update a running node, so the first step is to start the local node with the current runtime.

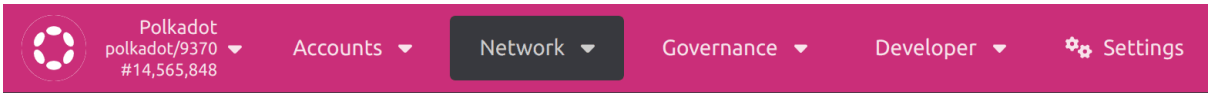
To start the node with the current runtime:

1. Open a terminal shell on your computer.
2. Change to the root directory where you compiled the Substrate node template.
3. Start the previously-compiled local node in development mode by running the following command:

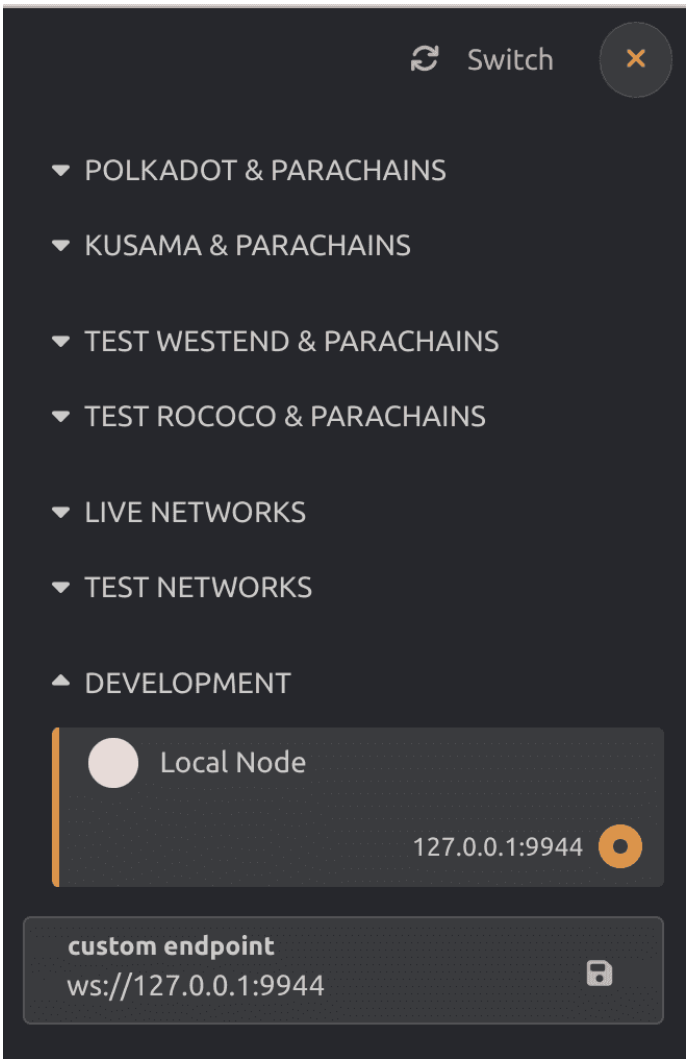
```
cargo run --release -- --dev
```

Leave this node running. You can edit and re-compile to upgrade the runtime without stopping or restarting the running node.

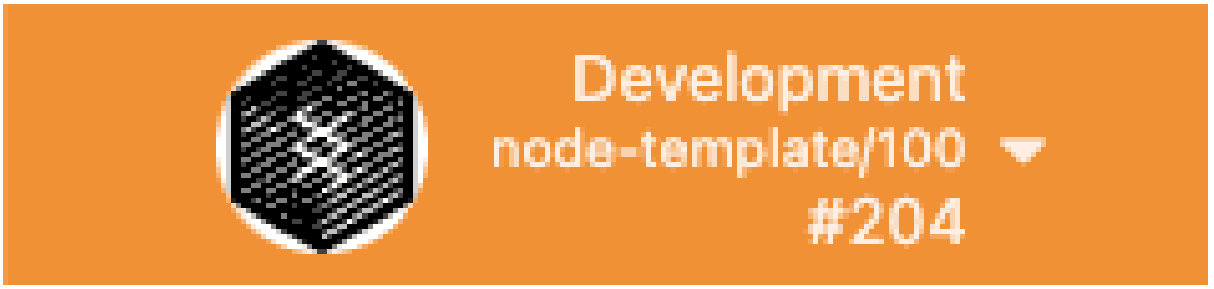
4. Open the Polkadot/Substrate Portal in a browser and connect to the local node.
5. Click the left-most dropdown menu to select the network.



6. Under **Development**, select **Local Node**, then click **Switch**.



7. In the upper left, notice the node template version is the default version 100.



Add the Utility pallet to the runtime dependencies

To update the dependencies for the runtime to include the Utility pallet:

1. Open a second terminal shell window or tab.
2. Change to the root directory where you compiled the Substrate node template.
3. Open the `runtime/Cargo.toml` file in a text editor.

Locate the `[dependencies]` section.

For example:

```
 [dependencies]
codec = { package = "parity-scale-codec", version = "3.0.0",
default-features = false, features = ["derive"] }
scale-info = { version = "2.1.1", default-features = false,
features = ["derive"] }

pallet-aura = { version = "4.0.0-dev", default-features = false,
git = "https://github.com/paritytech/substrate.git", branch =
"polkadot-v0.9.37" }
```

Add the Utility pallet as a dependency.

For example, add a single line with the following fields:

```
pallet-utility = {
  version = "4.0.0-dev",
  default-features = false,
  git = "https://github.com/paritytech/substrate.git",
  branch = "polkadot-v0.9.37"
}
```

Locate the `[features]` section and the list of the default features for the standard binary.

For example:

```
 [features]
default = ["std"]
std = [
  "frame-try-runtime?/std",
  "frame-system-benchmarking?/std",
  "frame-benchmarking?/std",
  "codec/std",
  "scale-info/std",
```

4. Add the Utility pallet to the list.

```
 "pallet-utility/std",
```

Save your changes and close the `Cargo.toml` file.

Add the Utility pallet configuration

To add the Utility types and configuration trait:

\Open the `runtime/src/lib.rs` file in a text editor.

Add the implementation for the Config trait for the Utility pallet.

```
 impl pallet_utility::Config for Runtime {
  type RuntimeEvent = RuntimeEvent;
  type RuntimeCall = RuntimeCall;
  type PalletsOrigin = OriginCaller;
  type WeightInfo =
pallet_utility::weights::SubstrateWeight<Runtime>;
}
```

Locate the `construct_runtime!` macro.

```
construct_runtime!(
 pub struct Runtime
 where
    Block = Block,
    NodeBlock = opaque::Block,
    UncheckedExtrinsic = UncheckedExtrinsic
 {
        System: frame_system,
        RandomnessCollectiveFlip:
pallet_randomness_collective_flip,
        Timestamp: pallet_timestamp,
        Aura: pallet_aura,
```

Add the Utility pallet inside the `construct_runtime!` macro.

```
Utility: pallet_utility,
```

Locate the `runtime_version` macro.

```
 #[sp_version::runtime_version]
pub const VERSION: RuntimeVersion = RuntimeVersion {
    spec_name: create_runtime_str!("node-template"),
    impl_name: create_runtime_str!("node-template"),
    authoring_version: 1,
    spec_version: 100,
    impl_version: 1,
    apis: RUNTIME_API_VERSIONS,
    transaction_version: 1,
    state_version: 1,
};
```

Update the value for the EXISTENTIAL_DEPOSIT for the Balances pallet.

```
pub const EXISTENTIAL_DEPOSIT: u128 = 1000 // Update this value.
```

This change increases the minimum balance an account is required to have on deposit to be viewed as a valid active account. This change doesn't remove any

accounts with balances between 500 and 1000. Removing accounts would require a storage migration. For information about upgrading data storage, see storage migration

Increment the `spec_version` to specify the new runtime version.

```
spec_version: 101,  // Change the spec_version from 100 to 101
```

The fields for the `runtime_version` specify the following information:

- `spec_name` specifies the name of the runtime.
- `impl_name` specifies the name of the outer node client.
- `authoring_version` specifies the version for block authors.
- `spec_version` specifies the version of the runtime.
- `impl_version` specifies the version of the outer node client.
- `apis` specifies the list of supported APIs.
- `transaction_version` specifies the version of the dispatchable function interface.
- `state_version` specifies the version of the key-value trie data structure that the runtime uses.

To upgrade the runtime, you must *increase* the `spec_version`. For more information, see the FRAME System module and the `can_set_code` method. Save your changes and close the `runtime/src/lib.rs` file.

Recompile and connect to the local node

1. Verify that the local node continues to run in the first terminal.

In the second terminal where you updated the runtime `Cargo.toml` and `lib.rs` files, recompile the runtime by running the following command

```
cargo build --release --package node-template-runtime
```

The `--release` command-line option requires a longer compile time. However, it generates a smaller build artifact that is better suited for submitting to the blockchain network. Storage optimization is *critical* for any blockchain. With this

command, the build artifacts are output to the `target/release` directory. The WebAssembly build artifacts are in the `target/release/wbuild/node-template-runtime` directory. For example, you should see the following WebAssembly artifacts:

node_template_runtime.compact.compressed.wasm

Node_template_runtime.compact.wasm

node_template_runtime.wasm

Execute a runtime upgrade

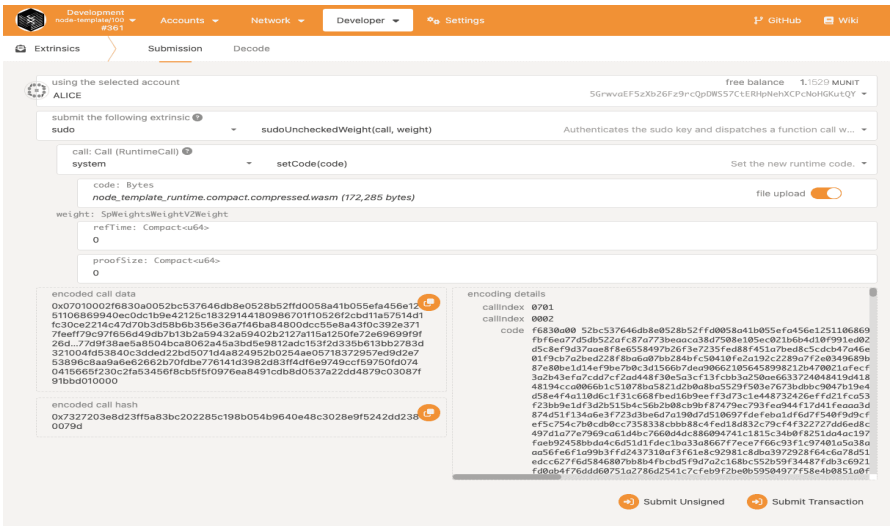
You now have a WebAssembly artifact that describes the modified runtime logic. However, the running node isn't using the upgraded runtime yet. To complete the upgrade, you need to submit a transaction that updates the node to use the upgraded runtime.

To update the network with the upgraded runtime:

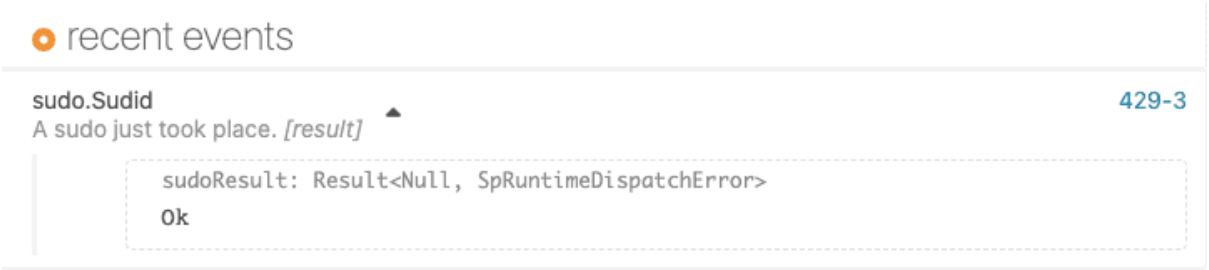
1. Open the Polkadot/Substrate Portal in a browser and connect to the local node.
2. Click **Developer** and select **Extrinsics** to submit a transaction for the runtime to use the new build artifact.
3. Select the administrative **Alice** account.
4. Select the **sudo** pallet and the **sudoUncheckedWeight(call, weight)** function.
5. Select **system** and **setCode(code)** as the call to make using the Alice account.
6. Click **file upload**, then select or drag and drop the compact and compressed WebAssembly file—`node_template_runtime.compact.compressed.wasm`—that you generated for the updated runtime.

   For example, navigate to the `target/release/wbuild/node-template-runtime` directory and select `node_template_runtime.compact.compressed.wasm` as the file to upload.

7. Leave both of the **weight** parameters set to the default value of `0`.



8. Click **Submit Transaction**.
9. Review the authorization, then click **Sign and Submit**.
10. Click **Network** and select **Explorer** to see that there has been a successful `sudo.Sudid` event.



After the transaction is included in a block, the node template version number indicates that the runtime version is now `101`. For example:

If your local node is producing blocks in the terminal that match what is displayed in the browser, you have completed a successful runtime upgrade.

11. Click **Developer** and select **Extrinsics**. Click on *submit the following extrinsic* and scroll to the bottom of the list. You will see **utility** as an option.

Verify the constant value by querying the chain state in the [Polkadot/Substrate Portal].

- Click **Developer** and select **Chain state**.
- Click **Constants**.
- Select the **balances** pallet.
- Select **existentialDeposit** as the constant value as the value to query.

Day-5

Adding a custom pallet to blockchain

As you saw in Build a local blockchain, the Substrate node template provides a working runtime that includes some default FRAME development modules—pallets—to get you started building a custom blockchain.

This module introduces the basic steps for adding a new pallet to the runtime for the node template. The steps are similar any time you want to add a new FRAME pallet to the runtime. However, each pallet requires specific configuration settings—for example, the specific parameters and types required to perform the functions that the pallet implements. For this tutorial, you'll add the Nicks pallet to the runtime for the node template, so you'll see how to configure the settings that are specific to the Nicks pallet. The Nicks pallet allows blockchain users to pay a deposit to reserve a nickname for an account they control. It implements the following functions:

- The set_name function to collect a deposit and set the name of an account if the name is not already taken.
- The clear_name function to remove the name associated with an account and return the deposit.
- The kill_name function to forcibly remove an account name without returning the deposit.

Note that this tutorial is a stepping stone to more advanced tutorials that illustrate how to add pallets with more complex configuration settings, how to create custom pallets, and how to publish pallets.

Before you begin

Before you begin, verify the following:

- You have configured your environment for Substrate development by installing Rust and the Rust toolchain.

- You have completed the Build a local blockchain tutorial and have the Substrate node template from the Developer Hub installed locally.
- You are generally familiar with software development and using command-line interfaces.
- You are generally familiar with blockchains and smart contract platforms.

Module objectives

By completing this tutorial, you will use the Nicks pallet to accomplish the following objectives:

- Learn how to update runtime dependencies to include a new pallet.
- Learn how to configure a pallet-specific Rust trait.
- See changes to the runtime by interacting with the new pallet using the front-end template.

Add the Nicks pallet dependencies

Before you can use a new pallet, you must add some information about it to the configuration file that the compiler uses to build the runtime binary.

For Rust programs, you use the Cargo.toml file to define the configuration settings and dependencies that determine what gets compiled in the resulting binary. Because the Substrate runtime compiles to both a native platform binary that includes standard library Rust functions and a WebAssembly (Wasm) binary that does not include the standard Rust library, the Cargo.toml file controls two important pieces of information:

- The pallets to be imported as dependencies for the runtime, including the location and version of the pallets to import.
- The features in each pallet that should be enabled when compiling the native Rust binary. By enabling the standard (std) feature set from each pallet, you can compile the runtime to include functions, types, and primitives that would otherwise be missing when you build the WebAssembly binary.

For information about adding dependencies in Cargo.toml files, see Dependencies in the Cargo documentation. For information about enabling and managing features from dependent packages, see Features in the Cargo documentation

To add the dependencies for the Nicks pallet to the runtime:

1. Open a terminal shell and change to the root directory for the node template.
2. Open the runtime/Cargo.toml configuration file in a text editor.
3. Locate the [dependencies] section and note how other pallets are imported.
4. Copy an existing pallet dependency description and replace the pallet name with pallet-nicks to make the pallet available to the node template runtime.

   For example, add a line similar to the following:

```
pallet-nicks = { version = "4.0.0-dev", default-features = false,
git = "https://github.com/paritytech/substrate.git", branch =
"polkadot-v0.9.37" }
```

5. This line imports the pallet-nicks crate as a dependency and specifies the following:
   - Version to identify which version of the crate you want to import.
   - The default behavior for including pallet features when compiling the runtime with the standard Rust libraries.
   - Repository location for retrieving the pallet-nicks crate.
   - Branch to use for retrieving the crate. Be sure to use the same version and branch information for the Nicks pallet as you see used for the other pallets included in the runtime.

6. These details should be the same for every pallet in any given version of the node template.

Add the pallet-nicks/std features to the list of features to enable when compiling the runtime.

```
[features]
default = ["std"]
std = [
  ...
  "pallet-aura/std",
  "pallet-balances/std",
  "pallet-nicks/std",
```

```
  ...
]
```

7. This section specifies the default feature set to compile for this runtime is the std features set. When the runtime is compiled using the std feature set, the std features from all of the pallets listed as dependencies are enabled. For more detailed information about how the runtime is compiled as a platform-native binary with the standard Rust library and as a WebAssembly binary using the no_std attribute, see Build process.

   If you forget to update the features section in the Cargo.toml file, you might see cannot find function errors when you compile the runtime binary.

8. Check that the new dependencies resolve correctly by running the following command

```
cargo check -p node-template-runtime --release
```

Review the configuration for Balances

Every pallet has a Rust trait called Config. The Config trait is used to identify the parameters and types that the pallet needs to carry out its functions.

Most of the pallet-specific code required to add a pallet is implemented using the Config trait. You can review what you to need to implement for any pallet by referring to its Rust documentation or the source code for the pallet. For example, to see what you need to implement for the nicks pallet, you can refer to the Rust documentation for pallet_nicks::Config or the trait definition in the Nicks pallet source code.

For this tutorial, you can see that the Config trait in the nicks pallet declares the following types:

```
pub trait Config: Config {
    type RuntimeEvent: From<Event<Self>> + IsType<<Self as
Config>::RuntimeEvent>;
```

```
    type Currency: ReservableCurrency<Self::AccountId>;
    type ReservationFee: Get<<<Self as Config>::Currency as
Currency<<Self as Config>::AccountId>>::Balance>;
    type Slashed: OnUnbalanced<<<Self as Config>::Currency as
Currency<<Self as Config>::AccountId>>::NegativeImbalance>;
    type ForceOrigin: EnsureOrigin<Self::RuntimeOrigin>;
    type MinLength: Get<u32>;
    type MaxLength: Get<u32>;
}
```

After you identify the types your pallet requires, you need to add code to the runtime to implement the Config trait. To see how to implement the Config trait for a pallet, let's use the Balances pallet as an example.

To review the Config trait for the Balances pallet:

1. Open the runtime/src/lib.rs file in a text editor.

Locate the Balances pallet and note that it consists of the following implementation (impl)code block:

```
pub type Balance = u128;

// ...

/// Existential deposit.
pub const EXISTENTIAL_DEPOSIT: u128 = 500;

impl pallet_balances::Config for Runtime {
  type MaxLocks = ConstU32<50>;
  type MaxReserves = ();
  type ReserveIdentifier = [u8; 8];
  /// The type for recording an account's balance.
  type Balance = Balance;
  /// The ubiquitous event type.
  type RuntimeEvent = RuntimeEvent;
  /// The empty value, (), is used to specify a no-op callback
function.
  type DustRemoval = ();
```

```
  /// Set the minimum balanced required for an account to exist
on-chain
  type ExistentialDeposit = ConstU128<EXISTENTIAL_DEPOSIT>;
  /// The FRAME runtime system is used to track the accounts that
hold balances.
  type AccountStore = System;
  /// Weight information is supplied to the Balances pallet by the
node template runtime.
  type WeightInfo =
pallet_balances::weights::SubstrateWeight<Runtime>;
}
```

As you can see in this example, the impl pallet_balances::Config block allows you to configure the types and parameters that are specified by the Balances pallet Config trait. For example, this impl block configures the Balances pallet to use the u128 type to track balances.

Implement the configuration for Nicks

Now that you have seen an example of how the Config trait is implemented for the Balances pallet, you're ready to implement the Config trait for the Nicks pallet.

To implement the nicks pallet in your runtime:

1. Open the runtime/src/lib.rs file in a text editor.
2. Locate the last line of the Balances code block.

Add the following code block for the Nicks pallet:

```
impl pallet_nicks::Config for Runtime {
// The Balances pallet implements the ReservableCurrency trait.
// `Balances` is defined in `construct_runtime!` macro.
type Currency = Balances;

// Set ReservationFee to a value.
type ReservationFee = ConstU128<100>;

// No action is taken when deposits are forfeited.
type Slashed = ();
```

```
// Configure the FRAME System Root origin as the Nick pallet
admin.
//
https://paritytech.github.io/substrate/master/frame_system/enum.Ra
wOrigin.html#variant.Root
type ForceOrigin = frame_system::EnsureRoot<AccountId>;

// Set MinLength of nick name to a desired value.
type MinLength = ConstU32<8>;

// Set MaxLength of nick name to a desired value.
type MaxLength = ConstU32<32>;

// The ubiquitous event type.
type RuntimeEvent = RuntimeEvent;
}
```

Add Nicks to the construct_runtime! macro.
For example:

```
construct_runtime!(
pub enum Runtime where
   Block = Block,
   NodeBlock = opaque::Block,
   UncheckedExtrinsic = UncheckedExtrinsic
 {
   /* --snip-- */
   Balances: pallet_balances,

   /*** Add This Line ***/
   Nicks: pallet_nicks,
 }
);
```

3. Save your changes and close the file.
4. Check that the new dependencies resolve correctly by running the following command:
   `cargo check -p node-template-runtime --release`
5. If there are no errors, you are ready to compile.

6. Compile the node in release mode by running the following command:
   cargo build --release

Start the blockchain node

After your node compiles, you are ready to start the node that has been enhanced with nickname capabilities from the Nicks pallet and interact with it using the front-end template.

To start the local Substrate node:

1. Open a terminal shell, if necessary.
2. Change to the root directory of the Substrate node template.
3. Start the node in development mode by running the following command:

```
./target/release/node-template --dev
```

4. In this case, the --dev option specifies that the node runs in developer mode using the predefined development chain specification. By default, this option also deletes all active data—such as keys, the blockchain database, and networking information—when you stop the node by pressing Control-c. Using the --dev option ensures that you have a clean working state any time you stop and restart the node.

5. Verify your node is up and running successfully by reviewing the output displayed in the terminal.
   If the number after finalized is increasing in the console output, your blockchain is producing new blocks and reaching consensus about the state they describe.

6. Keep the terminal that displays the node output open to continue.

Start the front-end template

Now that you have added a new pallet to your runtime, you can use the Substrate front-end template to interact with the node template and access the Nicks pallet.

To start the front-end template:

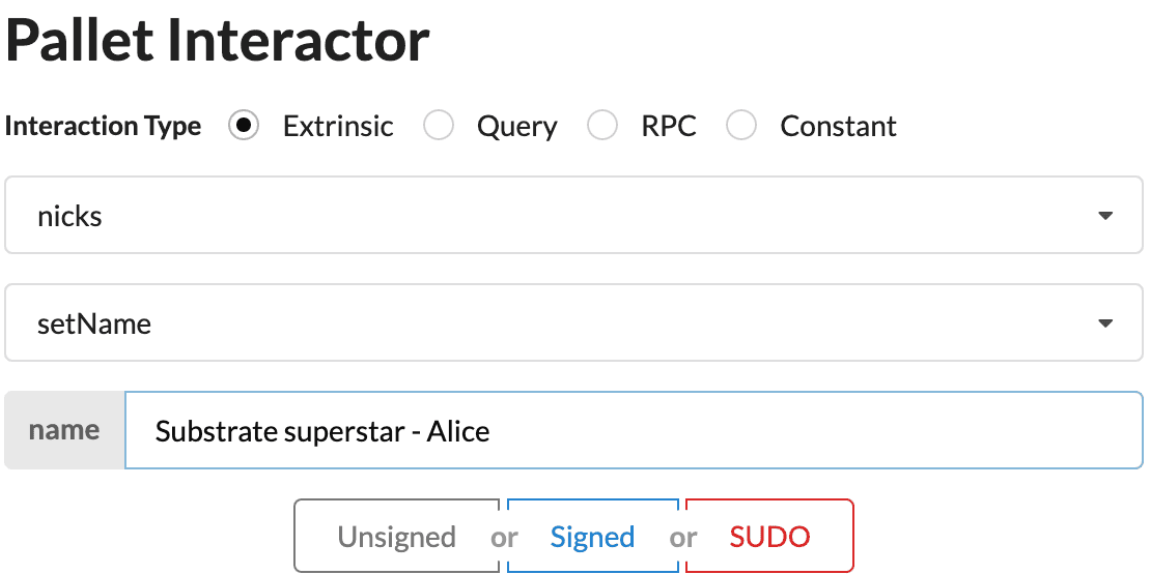
1. Open a new terminal shell on your computer.
2. In the new terminal, change to the root directory where you installed the front-end template.
3. Start the web server for the front-end template by running the following command:
   yarn start
4. Open http://localhost:8000/ in a browser to view the front-end template.

Set a nickname using the Nicks pallet

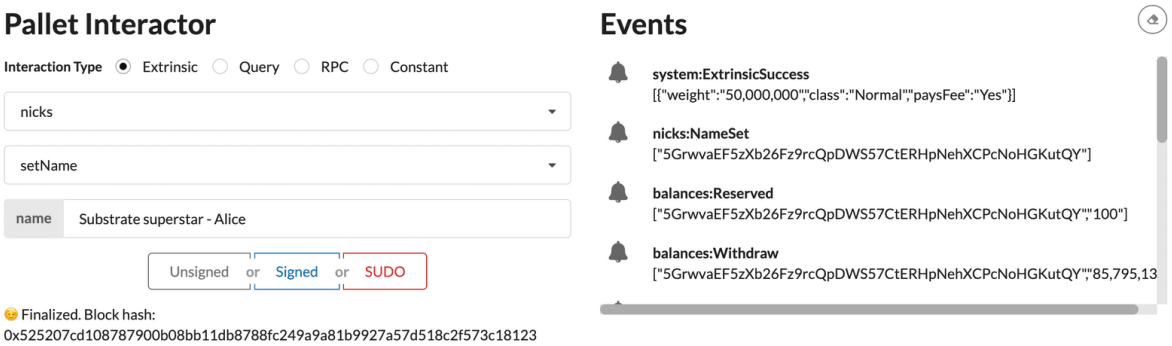
After you start the front-end template, you can use it to interact with the Nicks pallet you just added to the runtime.

To set a nickname for an account:

1. Check the account selection list to verify that the Alice account is currently selected.
2. In the Pallet Interactor component, verify that Extrinsic is selected.
3. Select nicks from the list of pallets available to call.
4. Select setName as the function to call from the nicks pallet.
5. Type a name that is longer than the MinNickLength (8 characters) and no longer than the MaxNickLength (32 characters).



6. Click Signed to execute the function.
7. Observe the status of the call change from Ready to InBlock to Finalized and the note the events emitted by the Nicks pallet.

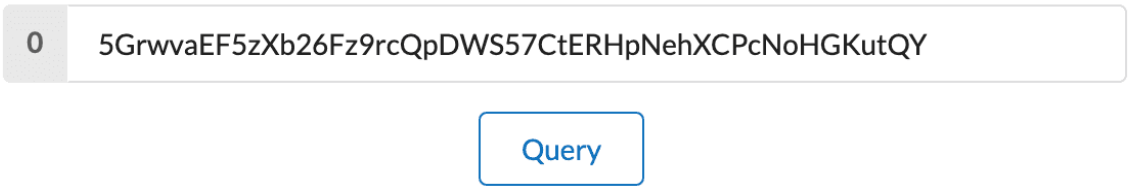


Query information for an account using the Nicks pallet

Next, you can use Query capability to read the value of Alice's nickname from the runtime storage for the Nicks pallet.

To return the information stored for Alice:

1. In the Pallet Interactor component, select Query as the Interaction Type.
2. Select nicks from the list of pallets available to query.
3. Select nameOf as the function to call.
4. Copy and paste the address for the alice account in the AccountId field, then click Query.



The return type is a tuple that contains two values:

- The hex-encoded nickname for the Alice account 537562737472617465207375706572737461722 02d20416c696365. If you convert the hex-encoded value to a string, you'll see the name you specified for the setName function.
- The amount that was reserved from Alice's account to secure the nickname (100).

5. If you were to query the Nicks pallet for the nameOf for Bob's account, you would see the value None returned because Bob has not invoked the setName function to reserve a nickname.