Dev activity log: initial Substrate customizations

This is a detailed log of the development activities that were required for the Substrate client customization. We'd like this log to serve as guidance for new developers wishing to understand the process with the goal of implementing their own versions.

Starting point: Substrate Template

Download the minimal template.

This version of the node includes the most basic features of a Substrate node.

Rust setup & dependencies:

Check out Polkadot's official installation instructions to install Rust and the additional dependencies needed for your system. Usually, it will be necessary to add the `wasm32v1-none` target, and the `rust-src` component, both of which can be installed, for example in Linux by executing the following commands:

```
Shell
$ rustup target add wasm32v1-none --toolchain stable-x86_64-unknown-linux-gnu
$ rustup component add rust-src --toolchain stable-x86_64-unknown-linux-gnu
```

Understand Substrate.

Basic components

Runtime

The runtime represents the state transition function for a blockchain. In Polkadot SDK, the runtime is stored as a Wasm binary in the chain state. The Runtime is stored under a unique state key and can be modified during the execution of the state transition function.

Node (Client)

The node, also known as the client, is the core component responsible for executing the Wasm runtime and orchestrating various essential blockchain components. It ensures the correct execution of the state transition function and manages multiple critical subsystems, including:

- *Wasm execution:* Runs the blockchain runtime, which defines the state transition rules.
- *Database management:* Stores blockchain data.

- *Networking:* Facilitates peer-to-peer communication, block propagation, and transaction gossiping.
- *Transaction pool (Mempool):* Manages pending transactions before they are included in a block.
- *Consensus mechanism:* Ensures agreement on the blockchain state across nodes.
- *RPC services:* Provides external interfaces for applications and users to interact with the node.

Substrate customizations

Include Griffin and Grandpa

As part of the Substrate customizations that can be done, we can modify the ledger model and the storage structure. This is where Griffin comes in. Griffin is a Substrate-based clone of a Cardano node. It incorporates a simplified eUTxO ledger and hosts a virtual machine capable of executing Plutus scripts for app-logic. This setup provides the essential primitives and logic required for our appchain nodes.

Another modification to the minimal template is the choice of a consensus mechanism. For this example, we have chosen GRANDPA for the finality protocol. Block finality is obtained through consecutive rounds of voting by validator nodes.

In the Changes made to the template section, we'll explain the modifications made to the template to add Griffin and Grandpa, but for your own choice of ledger and consensus mechanism the modifications can be made similarly.

Griffin vs. Substrate

A common Substrate node uses an account model ledger and its runtime is built with FRAME pallets, which are runtime "building blocks" or modules. In contrast, Griffin is designed for the UTxO model, making it incompatible with FRAME, as pallets inherently assume an account-oriented underlying model. This imposes a restriction on the design and implementation, opposed to usual Substrate app-chains. For developers coming from the Cardano ecosystem though, Griffin provides a familiar environment. We can think about decentralized applications in terms of validators and UTxOs, with the advantage of having a completely modifiable starting point as well.

Another key difference between Griffin and other Polkadot-based projects is the chain specification and execution. In general, the chain specification holds the entire code for the node, which is the *genesis*, among other information that is used at boot time, and it is passed to the node executable as an argument. In Griffin, the node executable takes as argument a simple file that describes the initial set of UTxOs, and some other details.

The original Griffin is a bit outdated, so to use it we updated some dependencies and made some minor compatibility changes, which will be detailed below. We encourage the use of the ad-hoc version provided in this repository at [griffin-core].

Changes made to Griffin

As mentioned, the version of Griffin that we use for this project has some modifications compared to the original. Most of these changes are dependency upgrades, but below we'll go over other more interesting modifications:

- **Authorities set function**: we re-implemented the authorities setting function to utilize the EUTxO model. The new function reads the authorities list from a UTxO that is set in the Genesis. A more detailed explanation on how it works and how to use it can be found in the respective readme.
- **Griffin-RPC**: We extended the native node RPC with some queries to obtain UTxOs information through an output reference, an address, or an asset class. Moreover, we also added a method to submit a transaction in CBOR format. More information and usage examples can be found in the Griffin RPC readme.
- **Wallet**: The wallet was also improved on through the addition of new functionalities like the queries by address and asset. The `build-tx` command was also modified to take as input a whole json file, instead of many arguments for each component of the transaction.

Guide to Griffin

Types

Griffin is based on Cardano and the eUTxO model so it bears a lot of similarities, but there are some key differences that we will clarify here.

Addresses

Griffin addresses are ed25519 keys. Public keys are prefixed by `61` and script addresses are prefixed by `70`. For simplicity, addresses don't have a staking part.

Transactions

Transactions don't pay fees. Staking, governance and delegation fields are included in the transaction body, but their functionalities are not supported.

Wallet

Griffin provides a CLI wallet to interact with the node. This wallet has helpful commands:

- `show-all-outputs`: Displays every UTxO in the chain with brief details about the owner, the coins and value.

- `show-outputs-at`: Displays UTxOs owned by the provided address.
- `show-outputs-with-asset`: Displays UTxOs that contain a certain token in its value.
- `insert-key`: Inserts a key into the wallet's keystore.
- `generate-key`: Creates a new key and inserts it into the keystore. Also displays the details.
- `show-balance`: summarizes Value amounts for each address.
- `build-tx`: Transaction builder command, which takes as input a complete Griffin transaction in json. This transaction must be balanced manually.

More information can be found in the wallet [readme] and also there are some usage examples in the [examples folder].

Changes made to the template

> **Polkadot-SDK**: From this point on, we steer away from using `polkadot-sdk` as one big dependency. Instead, we pick and choose what we need from the Polkadot-SDK and set each as their own dependency. This might look more complicated in terms of mantaining but we pull each dependency from the regsitry and as long as we pull the same stable version for each package there should not be any conflicts. At the time of writing this we use the release `polkadot-stable2506-2`. Having clarified this, it is necessary to add the dependencies in all `Cargo.toml` files, and also to modify the imports where used. To reduce clutter we won't be mentioning these steps while talking about the modifications.

Firstly, copy the source code for `griffin-core`, `griffin-rpc`, `griffin-wallet` and `demo`. Then we have to add these packages as workspace members to the project manifest: add the paths to the packages under the `[members]` section.

```
11    members = [
12        "demo/authorities",
13        "griffin-core",
14        "griffin-rpc",
15        "node",
16        "runtime",
17        "wallet",
```

And add the packages as workspace dependencies, so that they can be used by other modules:

```
56    [workspace.dependencies]
57    griffin-core = { default-features = false, path = "griffin-core" }
58    griffin-partner-chains-runtime = { path = "./runtime", default-features = false }
59    griffin-rpc = { default-features = false, path = "griffin-rpc" }


125   demo-authorities = { path = "demo/authorities", default-features = false }
```

To integrate Griffin into the node these are the necessary modifications:

Runtime

We can change the name of the runtime here:

```
2     name = "griffin-partner-chains-runtime"
```

Add dependencies to the Cargo.toml:

```
11    [dependencies]
12    demo-authorities = { workspace = true }
13    griffin-core = { workspace = true }
```

Add a new genesis file that includes the information for the initial set of UTxOs and a `get_genesis_config` function to build the genesis in the runtime.

```
57  pub fn get_genesis_config(genesis_json: String) -> GenesisConfig {
58      let mut json_data: &str = GENESIS_DEFAULT_JSON;
59      if !genesis_json.is_empty() {
60          json_data = &genesis_json;
61      };
62
63      match serde_json::from_str(json_data) {
64          Err(e) => panic!("Error: {e}\nJSON data: {json_data}"),
65          Ok(v) => v,
66      }
67  }
```

In the runtime library:

- Import Griffin types for `Address`, `AssetName`, `Datum`, `Input` and `PolicyId`. These types will be used to implement the runtime apis necessary for Griffin RPC.
- Import `TransparentUTxOSet` from Griffin.
- Import `MILLI_SECS_PER_SLOT` from Griffin, which will be used to define the slot duration of the chain.
- Import `GenesisConfig` from Griffin's config builder.

```
14    use griffin_core::genesis::config_builder::GenesisConfig;
15    use griffin_core::types::{Address, AssetName, Input, PolicyId};
16    use griffin_core::utxo_set::TransparentUtxoSet;
17    use griffin_core::MILLI_SECS_PER_SLOT;
18    pub use opaque::SessionKeys;
```

- Import Authorities from `demo`, which holds custom `aura_authorities` and `grandpa_authorities` implementations.

```rust
use demo_authorities as Authorities;
```

- Define `SessionKeys` struct within `impl_opaque_keys` macro, with fields for `Aura` and `Grandpa` public keys.

```rust
pub mod opaque {
    use super::*;
    use sp_core::{ed25519, sr25519};

    // This part is necessary for generating session keys in the runtime
    impl_opaque_keys! {
        pub struct SessionKeys {
            pub aura: AuraAppPublic,
            pub grandpa: GrandpaAppPublic,
        }
    }
    impl From<(sr25519::Public, ed25519::Public)> for SessionKeys {
        fn from((aura, grandpa): (sr25519::Public, ed25519::Public)) -> Self {
            Self {
                aura: aura.into(),
                grandpa: grandpa.into(),
            }
        }
    }
    // Typically these are not implemented manually, but rather for the pallet associated with the
    // keys. Here we are not using the pallets, and these implementations are trivial, so we just
    // re-write them.
    pub struct AuraAppPublic;
    impl BoundToRuntimeAppPublic for AuraAppPublic {
        type Public = AuraId;
    }

    pub struct GrandpaAppPublic;
    impl BoundToRuntimeAppPublic for GrandpaAppPublic {
        type Public = sp_consensus_grandpa::AuthorityId;
    }
}
```

- Remove `genesis_config_presets` mod definitions, since we are using our custom genesis.
- Define `Transaction`, `Block`, `Executive` and `Output` using the imported types from Griffin.

```rust
pub type Transaction = griffin_core::types::Transaction;
pub type Block = griffin_core::types::Block;
pub type Executive = griffin_core::Executive;
pub type Output = griffin_core::types::Output;
```

- Declare `Runtime` struct without FRAME pallets.

```rust
#[derive(Encode, Decode, PartialEq, Eq, Clone, TypeInfo)]
pub struct Runtime;
```

- Remove all FRAME trait implementations for Runtime.

`impl_runtime_apis` macro

Runtime APIs are traits that are implemented in the runtime and provide both a runtime-side implementation and a client-side API for the node to interact with. To utilize Griffin we provide new implementations for the required traits.

- Core: use Griffin's `Executive::execute_block` and `Executive::open_block` in `execute_block` and `initialize_block` methods implementations.

```
124         impl apis::Core<Block> for Runtime {
125             fn version() -> RuntimeVersion {
126                 VERSION
127             }
128
129             fn execute_block(block: Block) {
130                 Executive::execute_block(block)
131             }
132
133             fn initialize_block(header: &<Block as BlockT>::Header) -> sp_runtime::ExtrinsicInclusionMod
134                 Executive::open_block(header)
135             }
136         }
```

- Metadata: use trivial implementation.

- BlockBuilder: call Griffin's `Executive::apply_extrinsic` and `Executive::close_block` in `apply_extrinsic` and `finalize_block` methods, and provide trivial implementations of `inherent_extrinsics` and `check_inherents` methods.

```
152         impl apis::BlockBuilder<Block> for Runtime {
153             fn apply_extrinsic(extrinsic: <Block as BlockT>::Extrinsic) -> ApplyExtrinsicResult {
154                 Executive::apply_extrinsic(extrinsic)
155             }
156
157             fn finalize_block() -> <Block as BlockT>::Header {
158                 Executive::close_block()
159             }
160
161             fn inherent_extrinsics(_data: sp_inherents::InherentData) -> Vec<<Block as BlockT>::Extrinsic> {
162                 Vec::new()
163             }
```

- TaggedTransactionQueue: use Griffin's `Executive::validate_transaction`.

```rust
impl apis::TaggedTransactionQueue<Block> for Runtime {
    fn validate_transaction(
        source: TransactionSource,
        tx: <Block as BlockT>::Extrinsic,
        block_hash: <Block as BlockT>::Hash,
    ) -> TransactionValidity {
        Executive::validate_transaction(source, tx, block_hash)
    }
}
```

- SessionKeys: use the `generate` and `decode_into_raw_public_keys` methods of our defined `SessionKeys` type in `generate_session_keys` and `decode_session_keys` methods implementations.

```rust
impl apis::SessionKeys<Block> for Runtime {
    fn generate_session_keys(seed: Option<Vec<u8>>) -> Vec<u8> {
        opaque::SessionKeys::generate(seed)
    }

    fn decode_session_keys(
        encoded: Vec<u8>,
    ) -> Option<Vec<(Vec<u8>, sp_core::crypto::KeyTypeId)>> {
        opaque::SessionKeys::decode_into_raw_public_keys(&encoded)
    }
}
```

- GenesisBuilder: use Griffin's `GriffinGenesisConfigBuilder::build` and `get_genesis_config` functions to implement `build_state` and `get_preset` methods. Give trivial implementation of `preset_names`.

```rust
impl apis::GenesisBuilder<Block> for Runtime {
    fn build_state(config: Vec<u8>) -> sp_genesis_builder::Result {
        let genesis_config = serde_json::from_slice::<GenesisConfig>(config.as_slice())
            .map_err(|_| "The input JSON is not a valid genesis configuration.")?;

        griffin_core::genesis::GriffinGenesisConfigBuilder::build(genesis_config)
    }

    fn get_preset(_id: &Option<sp_genesis_builder::PresetId>) -> Option<Vec<u8>> {
        let genesis_config : &GenesisConfig = &genesis::get_genesis_config("".to_string());
        Some(serde_json::to_vec(genesis_config)
             .expect("Genesis configuration is valid."))
    }

    fn preset_names() -> Vec<sp_genesis_builder::PresetId> {
        vec![]
    }
}
```

- Include `sp_consensus_aura::AuraApi<Block, AuraId>` . Use custom `aura_authorities` implementation for `authorities` method. Use

`SlotDuration::from_millis` from `sp_consensus_aura` with previously imported MILLI_SECS_PER_SLOT to define `slot_duration`.

```
195     impl apis::AuraApi<Block, AuraId> for Runtime {
196         fn slot_duration() -> sp_consensus_aura::SlotDuration {
197             sp_consensus_aura::SlotDuration::from_millis(MILLI_SECS_PER_SLOT.into())
198         }
199
200         fn authorities() -> Vec<AuraId> {
201             Authorities::aura_authorities()
202         }
203     }
```

- Include `sp_consensus_grandpa::GrandpaApi<Block>`. Use custom `grandpa_authorities` implementation for the homonymous function from the api. Give a trivial implementation for `current_set_id`, `submit_report_equivocation_unsigned_extrinsic` and `generate_key_ownership_proof`.

```
205     impl apis::GrandpaApi<Block> for Runtime {
206         fn grandpa_authorities() -> sp_consensus_grandpa::AuthorityList {
207             Authorities::grandpa_authorities()
208         }
209
210         fn current_set_id() -> sp_consensus_grandpa::SetId {
211             0u64
212         }
213
214         fn submit_report_equivocation_unsigned_extrinsic(
215             _equivocation_proof: sp_consensus_grandpa::EquivocationProof<
216                 <Block as BlockT>::Hash,
217                 sp_runtime::traits::NumberFor<Block>,
218             >,
219             _key_owner_proof: sp_consensus_grandpa::OpaqueKeyOwnershipProof,
220         ) -> Option<()> {
221             None
222         }
223
224         fn generate_key_ownership_proof(
225             _set_id: sp_consensus_grandpa::SetId,
226             _authority_id: sp_consensus_grandpa::AuthorityId,
227         ) -> Option<sp_consensus_grandpa::OpaqueKeyOwnershipProof> {
228             None
229         }
230     }
```

- Include `griffin_core::utxo_set::TransparentUtxoSetApi<Block>`. Use `peek_utxo`, `peek_utxo_from_address` and `peek_utxo_with_asset` from `TransparentUtxoSet` to implement `peek_utxo`, `peek_utxo_by_address` and `peek_utxo_with_asset` from the api, respectively.

```
109         impl griffin_core::utxo_set::TransparentUtxoSetApi<Block> for Runtime {
110             fn peek_utxo(input: &Input) -> Option<Output> {
111                 TransparentUtxoSet::peek_utxo(input)
112             }
113
114             fn peek_utxo_by_address(addr: &Address) -> Vec<Output> {
115                 TransparentUtxoSet::peek_utxos_from_address(addr)
116             }
117
118             fn peek_utxo_with_asset(asset_name: &AssetName, asset_policy: &PolicyId) -> Vec<Output> {
119                 TransparentUtxoSet::peek_utxos_with_asset(asset_name, asset_policy)
120             }
121         }
```

- Remove `OffchainWorkerApi`, `AccountNonceApi` and `TransactionPaymentApi` trait implementations.

Node

Add dependencies to the Cargo.toml:

```
20  griffin-core = { workspace = true }
21  griffin-partner-chains-runtime = { workspace = true }
22  griffin-rpc = { workspace = true }
```

In chain_spec, we redefine the functions that build the chain from the specification:

- Import `get_genesis_config` from runtime genesis, and `WASM_BINARY` from runtime.

```
1  use griffin_partner_chains_runtime::genesis::get_genesis_config;
2  use griffin_partner_chains_runtime::WASM_BINARY;
```

- Modify `development_chain_spec()` to take a String as an argument and add the logic that uses it. The name was changed to reflect the purpose of the function more accurately.

```
8  pub fn development_config(genesis_json: String) -> Result<ChainSpec, String> {
9      Ok(ChainSpec::builder(
10         WASM_BINARY.ok_or_else(|| "Development wasm not available".to_string())?,
11         None,
12     )
13     .with_name("Development")
14     .with_id("dev")
15     .with_chain_type(ChainType::Development)
16     .with_genesis_config_patch(serde_json::json!(get_genesis_config(genesis_json)))
17     .build())
18 }
```

- Add a new function for the configuration of a local test chain.

```rust
pub fn local_testnet_config(genesis_json: String) -> Result<ChainSpec, String> {
    Ok(ChainSpec::builder(
        WASM_BINARY.ok_or_else(|| "Development wasm not available".to_string())?,
        None,
    )
    .with_name("Local Testnet")
    .with_id("local_testnet")
    .with_chain_type(ChainType::Local)
    .with_genesis_config_patch(serde_json::json!(get_genesis_config(genesis_json)))
    .build())
}
```

In cli:

- Add new `ExportChainSpec` command and add deprecated warning to `BuildSpec` command.

```rust
    #[deprecated(
        note = "build-spec command will be removed after 1/04/2026. Use export-chain-spec command instead"
    )]
    BuildSpec(sc_cli::BuildSpecCmd),

    /// Export the chain specification.
    ExportChainSpec(sc_cli::ExportChainSpecCmd),
```

In command:

- Modify chain name

```rust
    fn impl_name() -> String {
        "Griffin solochain node based on Substrate / Polkadot SDK".into()
    }
```

- Modify `load_spec` function to use the new config functions defined in `chain_spec.rs`.

```rust
    fn load_spec(&self, id: &str) -> Result<Box<dyn sc_service::ChainSpec>, String> {
        Ok(match id {
            "dev" => Box::new(chain_spec::development_config("".to_string())?),
            "" | "local" => Box::new(chain_spec::local_testnet_config("".to_string())?),
            path => {
                let file_content =
                    std::fs::read_to_string(path).expect("Unable to read the initialization file");
                Box::new(chain_spec::local_testnet_config(file_content)?)
            }
        })
    }
```

- Add new `ExportChainSpec` command and a deprecated warning to `BuildSpec`.

```
70          Some(Subcommand::ExportChainSpec(cmd)) => {
71              let chain_spec = cli.load_spec(&cmd.chain)?;
72              cmd.run(chain_spec)
73          }
```

```
53          #[allow(deprecated)]
54          Some(Subcommand::BuildSpec(cmd)) => {
55              let runner = cli.create_runner(cmd)?;
56              runner.sync_run(|config| cmd.run(config.chain_spec, config.network))
57          }
```

- Provide Griffin's `OpaqueBlock` type in `NetworkWorker`.

```
132                     sc_network::config::NetworkBackendType::Libp2p => service::new_full::<
133                         sc_network::NetworkWorker<
134                             griffin_core::types::OpaqueBlock,
135                             <griffin_core::types::OpaqueBlock as sp_runtime::traits::Block>::Hash,
136                         >,
137                     >(
```

In service:

- Import `GriffinGenesisBlockBuilder` and `OpaqueBlock` as `Block` from Griffin.
- Import `self` and `RuntimeApi` from our runtime (necessary if the runtime name changed).

```
4   use griffin_core::{genesis::GriffinGenesisBlockBuilder, types::OpaqueBlock as Block};
5   use griffin_partner_chains_runtime::{self, RuntimeApi};
```

- Within `new_partial`:
    - Define a new backend using `sc_service::new_db_backend`.

```
51          let backend = sc_service::new_db_backend(config.db_config())?;
```

    - Define `genesis_block_builder` from `GriffinGenesisBlockBuilder`.

```
52          let genesis_block_builder = GriffinGenesisBlockBuilder::new(
53              config.chain_spec.as_storage_builder(),
54              !config.no_genesis(),
55              backend.clone(),
56              executor.clone(),
57          )?;
```

- Modify the creation of the initial parts of the node to use our custom genesis block builder.

```
59          let (client, backend, keystore_container, task_manager) =
60              sc_service::new_full_parts_with_genesis_builder::<Block, RuntimeApi, _, _>(
61                  config,
62                  telemetry.as_ref().map(|(_, telemetry)| telemetry.handle()),
63                  executor,
64                  backend,
65                  genesis_block_builder,
66                  false,
67              )?;
```

  - Delete `offchain_worker` definition, as Griffin's executive module doesn't implement it.

- Within `new_full`:

  - Define `chain_spec` and its new way of parsing the json.

```
168         let chain_spec =
169             &serde_json::from_str::<serde_json::Value>(&config.chain_spec.as_json(false).unwrap())
170                 .unwrap();
```

  - Define `zero_time` for the ledger.

```
171         let zero_time = chain_spec["genesis"]["runtimeGenesis"]["patch"]["zero_time"]
172             .as_u64()
173             .unwrap();
```

  - Sleep until reaching zero time for the genesis of the chain.

```
204         let now = SystemTime::now()
205             .duration_since(UNIX_EPOCH)
206             .unwrap()
207             .as_millis() as u64;
208
209         // Wait until genesis time
210         sleep(Duration::from_millis(
211             zero_time.checked_sub(now).unwrap_or(0),
212         ));
```

In rpc:

- Import `CardanoRpc` and `CardanoRpcApiServer` from Cardano RPC within Griffin RPC.
- Import `TransparentUtxoSetRpc` and `TransparentUtxoSetRpcApiServer` from RPC within Griffin RPC.

```
use griffin_rpc::cardano_rpc::{CardanoRpc, CardanoRpcApiServer};
use griffin_rpc::rpc::{TransparentUtxoSetRpc, TransparentUtxoSetRpcApiServer};
```

- Add TransparentUtxoSetApi dependency to `create_full` function.

```
    C::Api: griffin_core::utxo_set::TransparentUtxoSetApi<
        <P as sc_transaction_pool_api::TransactionPool>::Block,
    >,
```

- Add the new RPC modules in the `create_full` function.

```
    let mut module = RpcModule::new(());
    let FullDeps { client, pool } = deps;

    module.merge(CardanoRpc::new(client.clone(), pool.clone()).into_rpc())?;
    module.merge(TransparentUtxoSetRpc::new(client.clone()).into_rpc())?;

    Ok(module)
```

Troubleshooting

These are some common errors that can happen when developing on Substrate:

STD related issues

Errors like:

- `Double lang item in crate <crate> (which` std`/` serde `depends on):...`
- `Attempted to define built-in macro more than once`

happen commonly when using std crates in a non-std environment, like Substrate's runtime. Std crates can't be used because we compile to WASM. If you run into an error like this and the crate you are using is no-std, make sure you are setting them up correctly. For example, make sure that the dependency is imported with `default-features = false` or that the std feature is set correctly in the respective `Cargo.toml`. If you are writing a new module, make sure that it is premised by ´#![cfg_attr(not(feature = "std"), no_std)]´.

`Alloc` feature

When trying to use `alloc` features like `vec`, you might run into the trouble that the compiler can't find the `alloc` crate. This feature can be imported from various dependencies like `serde` and `serde_json`. To use it make sure to add `extern crate alloc;` at the top of your file.