parity

Hello? Yes, this is Runtime

Calling into the Runtime in a safe and easy way.

Bastian Köcher
Software developer, Parity Technologies
bastian@parity.io | @bkchr

Agenda

- What is Substrate/a Runtime/a Runtime API?
- Declaring a Runtime API
- Implementing a Runtime API
- Calling a Runtime API

What is Substrate?

Substrate is an **open source**, **modular**, and **extensible** framework for building blockchains.

What is Substrate?

Substrate provides all the core components of a Blockchain:

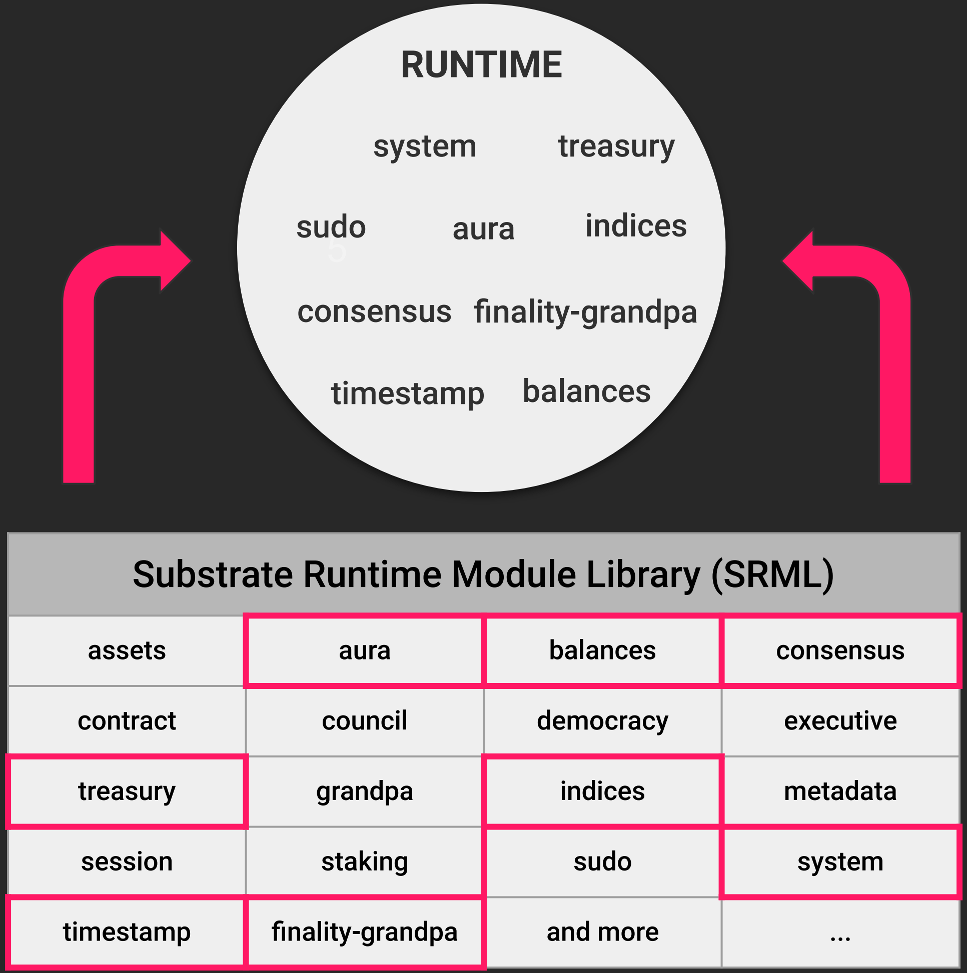
- Database Layer
- Networking Layer
- Consensus Engine
- Transaction Queue
- Library of Runtime Modules

Each of which can be customized and extended.

What is a Runtime?

The runtime is the **block execution logic** of the blockchain, i.e. the State Transition Function.

It is composed of **Runtime Modules**.



RUNTIME

system treasury

sudo aura indices

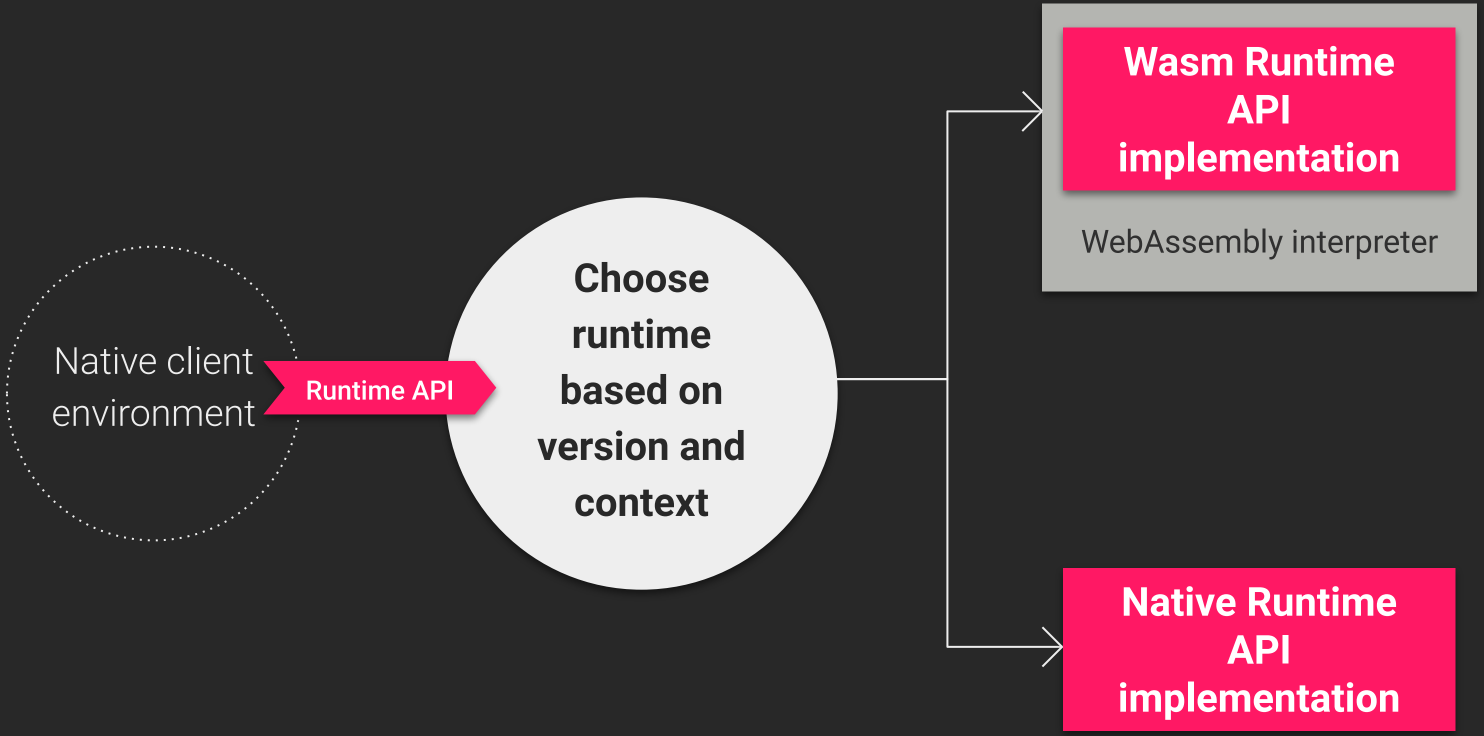
consensus finality-grandpa

timestamp balances

Substrate Runtime Module Library (SRML)

assets	aura	balances	consensus
contract	council	democracy	executive
treasury	grandpa	indices	metadata
session	staking	sudo	system
timestamp	finality-grandpa	and more	...

What is a Runtime API?


Native client environment
Runtime API
Choose runtime based on version and context
Wasm Runtime API implementation
WebAssembly interpreter
Native Runtime API implementation

What is a Runtime API?

A Runtime API is a well-defined interface between the native client and the wasm/native runtime.

- ***Core*** Substrate api:
  - ***version*** - get the version of your runtime
  - ***execute_block*** - execute all transactions and check that hashes are correct
  - ***initialize_block*** - initialize the runtime at the given block

Declaring a Runtime API

Declaring a Runtime API

```
decl_runtime_apis! {
    pub trait Hello {
        fn world() -> Vec<u8>;
    }
}
```

- Declaration is wrapped in a macro
- Expands to a client side and a runtime side declaration

Declaring a Runtime API

Client side

Declaring a Runtime API - Client side

```
pub trait Hello<Block: BlockT>: Core<Block> {
    fn world(&self, &BlockId<Block>) -> Result<Vec<u8>, Error>;

    fn world_with_context(
        &self, &BlockId<Block>, ExecutionContext
    ) -> Result<Vec<u8>, Error>;

    fn Hello_world_runtime_api_impl(
        &self, &BlockId<Block>, ExecutionContext, Option<()>, Vec<u8>
    ) -> Result<NativeOrEncoded<Vec<u8>>>;
}
```

Declaring a Runtime API - Client side

```rust
pub trait Hello {
    fn world() -> Vec<u8>;
}
```

Declaring a Runtime API - Client side

```rust
pub trait Hello<Block: BlockT> {
    fn world() -> Vec<u8>;
}
```

- ***Block*** generic parameter is added

Declaring a Runtime API - Client side

```
pub trait Hello<Block: BlockT>: Core<Block> {
    fn world() -> Vec<u8>;
}
```

- ***Block*** generic parameter is added
- ***Core*** trait is added as supertrait

Declaring a Runtime API - Client side

```rust
pub trait Hello<Block: BlockT>: Core<Block> {
    fn world(&self) -> Vec<u8>;
}
```

- ***Block*** generic parameter is added
- ***Core*** trait is added as supertrait

- ***self*** parameter is added

Declaring a Runtime API - Client side

```
pub trait Hello<Block: BlockT>: Core<Block> {
    fn world(&self, &BlockId<Block>) -> Vec<u8>;
}
```

- ***Block*** generic parameter is added
- ***Core*** trait is added as supertrait

- ***self*** parameter is added
- ***BlockId*** parameter is added

Declaring a Runtime API - Client side

```
pub trait Hello<Block: BlockT>: Core<Block> {
    fn world(&self, &BlockId<Block>) -> Result<Vec<u8>, Error>;
}
```

- ***Block*** generic parameter is added
- ***Core*** trait is added as supertrait

- ***self*** parameter is added
- ***BlockId*** parameter is added
- Return value is wrapped into a ***Result***

Declaring a Runtime API - Client side

```
pub trait Hello<Block: BlockT>: Core<Block> {
    fn world(&self, &BlockId<Block>) -> Result<Vec<u8>, Error>;

    fn world_with_context(
        &self, &BlockId<Block>, ExecutionContext
    ) -> Result<Vec<u8>, Error>;

    fn Hello_world_runtime_api_impl(
        &self, &BlockId<Block>, ExecutionContext, Option<()>, Vec<u8>
    ) -> Result<NativeOrEncoded<Vec<u8>>>;
}
```

Declaring a Runtime API

Runtime side

Declaring a Runtime API - Runtime side

```
pub trait Hello<Block: BlockT> {
    fn world() -> Vec<u8>;
}
```

- Same declaration as given to the macro
- ***Block*** generic parameter is added as well
- Is hidden in a module

Implementing a Runtime API

Implementing a Runtime API

```
impl_runtime_apis! {
    impl api::Hello<Block> for Runtime {
        fn world() -> Vec<u8> {
            "Hello World".encode()
        }
    }
}
```

Implementing a Runtime API

Client side

Implementing a Runtime API - Client side

```rust
pub struct RuntimeApi {}

pub struct RuntimeApiImpl {}
```

- ***RuntimeApi*** implements ***ConstructRuntime***
- ***RuntimeApiImpl*** implements all given traits

Implementing a Runtime API

Runtime side

Implementing a Runtime API - Runtime side

```rust
impl api::runtime_decl_for_Hello::Hello<Block> for Runtime {
    fn world() -> Vec<u8> {
        "Hello World".encode()
    }
}
```

- Implements the trait for the ***Runtime***

Implementing a Runtime API - Runtime side

```
const RUNTIME_API_VERSIONS: ApisVec = Cow::Borrowed(&[
    ( api::runtime_decl_for_Hello::ID, api::runtime_decl_for_Hello::VERSION ),
]);
```

- ***RUNTIME_API_VERSIONS*** - contains all API versions + IDs
- Is exposed by the runtime version to the client

Implementing a Runtime API - Runtime side

```
pub mod api {
    #[no_mangle]
    pub fn Hello_world(input_data: *mut u8, input_len: usize) -> u64 {
        ...
    }
}
```

- Expose a function in WASM per trait method
- Decodes input parameters and calls trait method
- Returns encoded result

Calling a Runtime API

Calling a Runtime API

```
let client = create_client();
let runtime_api = client.runtime_api();
let block_id = BlockId::Number(0);

if runtime_api.has_api::<Hello<Block>>(&block_id) {
    let res = runtime_api.world(&block_id).unwrap()
    println!("{}", String::decode(&mut &res[..]).unwrap());
}
```

Summary

- Declare your runtime api using ***decl_runtime_apis!***
- Declaration is created for the client and the runtime
- Client side expects target block

- Implement your runtime api using ***impl_runtime_apis!***
- Each trait method exposes a function in WASM
- Client side implementation is provided by ***RuntimeApi*** and ***RuntimeApiImpl***

Questions?

bastian@parity.io

@bkchr

Backup

Declaring a Runtime API - Client side

```
impl<Block: BlockT> RuntimeApiInfo for Hello<Block> {
    const ID: [u8; 8] = [60u8, 92u8, 138u8, 31u8, 219u8, 32u8, 104u8, 134u8];
    const VERSION: u32 = 1u32;
}
```

- ***ID*** - Hash of “Hello”
- ***VERSION*** - The version of the API.

Declaring a Runtime API - Attributes

```
decl_runtime_apis! {
    #[api_version(2)]
    pub trait Hello {
        #[renamed("hello_world", 1)]
        fn world() -> Vec<u8>;
        #[changed_in(2)]
        fn world(id: u32);
    }
}
```