
Kagome

Release 0.0.1

Soramitsu Co., Ltd.

Aug 05, 2021

TABLE OF CONTENTS

- 1 Kagome Core Development 3**
 - 1.1 Development 3
 - 1.2 CodeStyle 14
- 2 Overview 15**
 - 2.1 Getting started 15
- 3 Tutorials 17**
 - 3.1 Your first Kagome chain 17
 - 3.2 Start private Kagome network 20
- 4 Kagome in media 23**

Kagome is a C++ implementation of [Polkadot Host](#) developed by [Soramitsu](#) and funded by Web3 Foundation [grant](#).

WARNING: Kagome is early-stage software. While we are doing our best to be compatible with other Polkadot Host implementations, we cannot guarantee full compatibility due to specification being in development.

This documentation will guide you through the installation, deployment, and launch of Kagome network, and explain to you how to write an application for it.

If you would like to understand how different components of Kagome are implemented you are welcome to review [documentation generated from the source code](#).

KAGOME CORE DEVELOPMENT

1.1 Development

The information affects Core-Contributors who like to develop on Kagome itself.

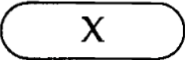
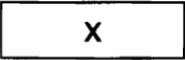
Ensure that you go through the following documentation before contributing:

1.1.1 Terms

1. **definition** - provides a unique description of an entity (type, instance, function) within a program:
 1. it declares a function without specifying its body
 2. it contains an `extern` specifier and no initializer or function body
 3. it is the declaration of a static class data member within a class definition
 4. it is a class name declaration
 5. it is a `typedef` or `using` declaration
2. **declaration** - introduces a name into a program:
 1. it defines a static class data member
 2. it defines a non-inline member function
3. **free function** - non-member function, which is not *friend*.
4. **translation unit** - single C++ source file after preprocessor finished including all of the header files.
5. **internal linkage** - name has internal linkage if it is local to its *translation unit* and cannot collide with an identical name defined in another translation unit at link time.
6. **external linkage** - name has external linkage if in multi-file program, that name can interact with other translation units at link time.
7. **target** - shared or static library, executable or command, added to CMake with `add_library`, `add_executable`, `add_custom_target`. With make executed as `make <target>`.
8. **public API** - is an interface which is programmatically accessible or detectable by a client.
9. **component** - is the smallest unit of physical design. Or, a set of *translation units*, compiled in a single *target* as a single *library* or *executable*.
10. **unit test** - a set of *translation units* intended to test *public API* of a specific *component* (exactly one!).

11. **regression test** - refers to the practice of comparing the results of running a program given a specific input with a fixed set of expected results, in order to verify that the program continues to behave as expected from one version to the next. In other words, tests which persist across versions.
12. **integration test** - refers to the practice when group of *components* (or *packages*) is grouped and tested together as a single unit. Example: session manager with postgres database.
13. **white-box testing** - refers to the practice of verifying the expected behavior of a component by exploiting knowledge of its underlying implementation.
14. **black-box testing** - refers to the practice of verifying the expected behavior of a component based solely on its specification (without knowledge of its implementation).
15. **protocol class** - an abstract class is a protocol class if:
 1. it neither contains nor inherits from classes that contain member data, non-virtual functions, or private (protected) members of any kind.
 2. it has a non-inline virtual destructor defined with an empty implementation (default implementation).
 3. all member functions other than the destructor including inherited functions, are declared pure virtual and left undefined.
16. **packages** - is a collection of *components* organized as a physically cohesive unit.
17. **package group** - is a collection of *packages* organized as a physically cohesive unit.
18. **library** - is a collection of *package groups* organized as a physically cohesive unit.

Levelization

<u>DEFINITION:</u>	
<u>NOTATION</u>	<u>MEANING</u>
	X is a logical entity (e.g., class).
	x is a physical entity (e.g., file).
$B \xrightarrow{\text{IsA}} A$	B is a kind of A.
$B \circ \xrightarrow{\text{Uses-In-The-Interface}} A$	B uses A in B's interface.
$B \bullet \xrightarrow{\text{Uses-In-The-Implementation}} A$	B uses A in B's implementation.

Given notation as above, we can build a physical (files) and logical (entities) dependency graph.

In perfectly testable system in such graphs there are no cycles (in C++ terms - no circular dependencies) and system consists of clearly defined components, which may be combined in packages.

This graph then is topologically sorted – this helps to visualize and levelize components.

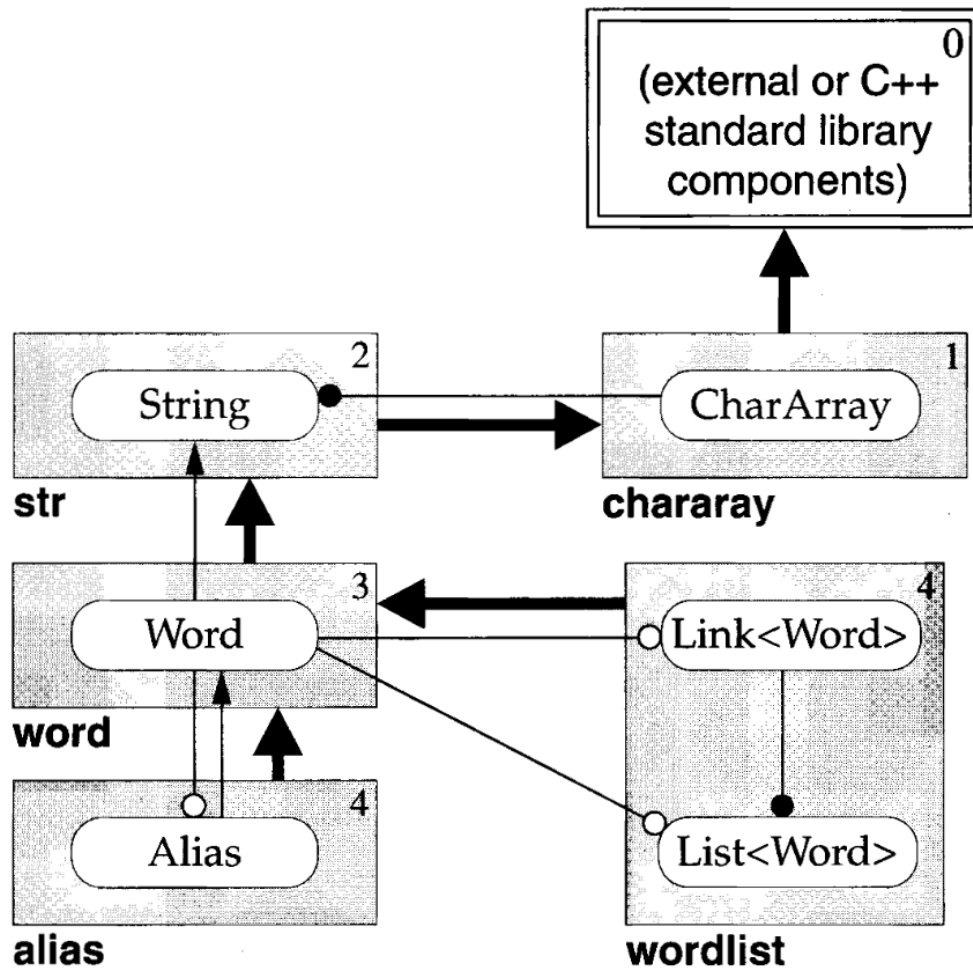


Figure 4-11: Levelized Component Dependency Diagram

1.1.2 Rules

We define categories of rules:

- **Design rules** - recommendations that flatly proscribe or require a given practice without exception.
- **Guidelines** - suggested practices of a more abstract nature for which exceptions are sometimes legitimately made.
- **Principles** - certain observations and truths that have often proved useful during the design process but must be evaluated in the context of a specific design.

All of these rules are described in the [^1], readers are welcomed *not to ask rationale here*, but search through the book by themselves for answers.

All used terms are described in [terms.md](#).

1. **major design rule** - It is almost always an error to place a *definition* with *external linkage* in a `.h` file.

```
// radio.h
#ifndef INCLUDED_RADIO
#define INCLUDED_RADIO
int z; // illegal: external data definition
extern int LENGTH = 10; // illegal: external data definition
const int WIDTH = 5; // avoid: constant data definition
static int y; // avoid: static data definition
static void func() {...} // avoid: static function definition
class Radio {
    static int s_count; // fine: static member declaration
    static const double S_PI; // fine: static const member dec.
    int d_size; // fine: member data definition
public:
    int size() const; // fine: member function declaration
}; // fine: class definition

inline int Radio::size() const {
    return d_size;
} // fine: inline function definition

int Radio::s_count; // illegal: static member definition

double Radio::-S_PI = 3.14159265358; // illegal: static const member def,
int Radio::size() const { /*...*/ } // illegal: member function definition
#endif
```

2. **major design rule** - avoid *free functions* with *external linkage*. The *definitions* to be avoided at file scope in `.c` files are data and functions that have *not* been declared static [^1: 1.1.4]

```
// file1.c
int i; // avoid: external linkage
int max(int a, int b){...} // avoid: external linkage
inline int min(){...} // fine: internal linkage
static int mean(){...} // fine: internal linkage
class Link; // fine: internal linkage
enum {...} // fine: internal linkage
const double PI = 3; // fine: internal linkage
static const char *names[] = {"a", "b"} // fine: internal linkage
typedef struct {...} mytype; // fine: does not introduce new type.
```

3. **major design rule** - keep class data members private.
4. **major design rule** - avoid *free functions* (except operator functions) at file scope in `.h`.
5. **major design rule** - avoid enums, typedefs and constants at file scope in `.h` files.
6. **major design rule** - avoid using preprocessor macros in header files except as include guards.
7. **major design rule** - only classes, structures, unions and free operator functions should be *declared* at file scope in `.h` file. Only classes, structures, unions, and inline (member or free operator) functions should be *defined* at file scope in a `.h` file.
8. **major design rule** - place a unique and predictable (internal) include guard around the contents of each header file.
9. **guideline** - document the *public API* so that they are usable by others. Have at least one other developer review each interface.

10. **guideline** - explicitly state conditions under which behavior is undefined.
11. **principle** - the use of `assert` statements can help to document the assumptions you make when implementing your code.
12. **principle** - a component is the appropriate fundamental unit of design.
13. **major design rule** - logical entities *declared* within a component should not be *defined* outside that component.
14. **minor design rule** - the root names of the `.cpp` and the `.hpp` file that comprise a component should match exactly.
15. **major design rule** - the `.cpp` file of every component should include its own `.hpp` file as the first substantive line of code.
16. **guideline** - clients should include header files providing required type definitions directly; except for non-private inheritance, avoid relying on one header file to include another.
17. **major design rule** - avoid definitions with *external linkage* in the `.cpp` file of a component that are not declared explicitly in the corresponding `.hpp` file.
18. **major design rule** - avoid accessing a *definition* with *external linkage* in another component via a local declaration; instead, include the `.hpp` file for that component.
19. **guideline** - a component X should include `y.hpp` only if X makes direct substantive use of a class or free operator function defined in Y.
20. **principle** - granting (local) friendship to classes defined within the same component does not violate encapsulation.

Example: defining an iterator **class along with** a container **class in** the same `␣`
 ↪ component enables user extensibility, improves maintainability **and** enhances `␣`
 ↪ reusability **while** preserving encapsulation.

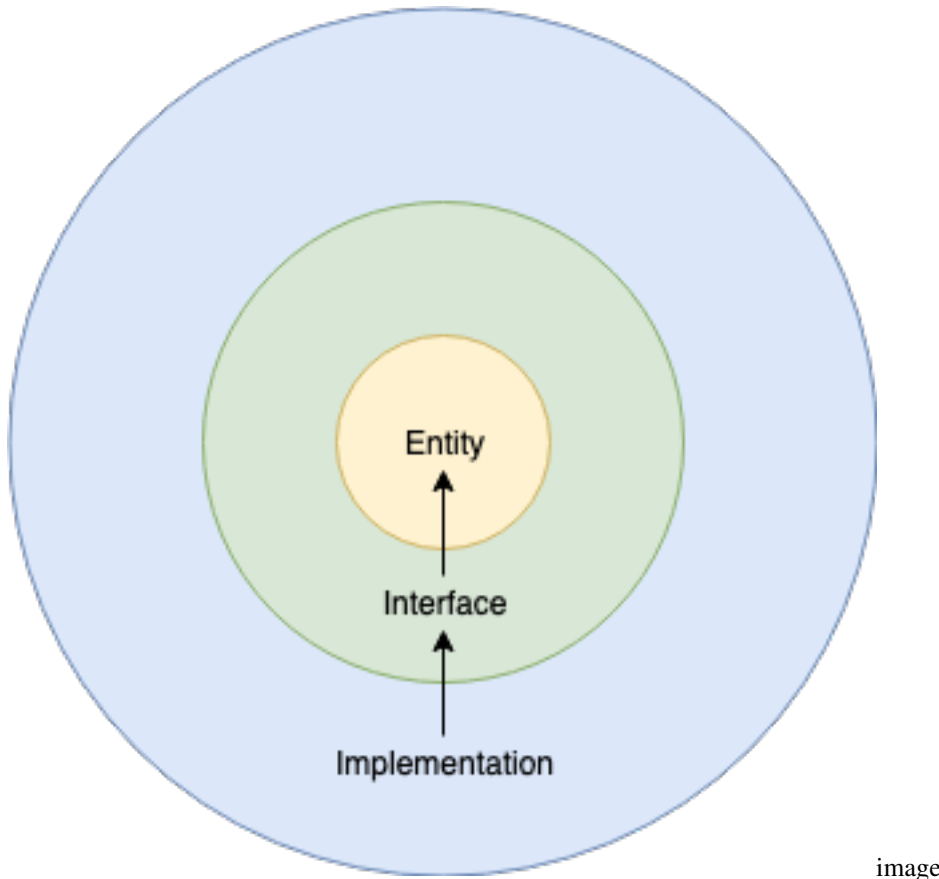
1. **principle** - all tests must be done in isolation. Testing a component in isolation is an effective way to ensure reliability.
2. **principle** - every directed acyclic graph can be assigned unique level numbers; a graph with cycles cannot. A physical dependency graph that can be assigned unique level numbers is said to be *levelizable*.
3. **principle** - in most real-world situations, large designs must be levelizable if they are able to be tested effectively.
4. **principle** - testing only the functionality *directly implemented* within a component enables the complexity of the test to be proportional to the complexity of the component.
5. **guideline** - avoid cyclic physical dependencies among dependencies.
6. **principle** - factoring a concrete class into two classes containing higher and lower levels of functionality can facilitate levelization.
7. **principle** - factoring an abstract base class into two classes - one defining a pure interface, the other defining its partial implementation - can facilitate levelization.
8. **principle** - a *protocol class* can be used to eliminate both compile and link time dependencies.
9. **major design rule** - prepend every global identifier with its package prefix.
10. **major design rule** - avoid cyclic dependencies among packages.
11. **guideline** - avoid declaring results returned by value from functions as `const`.
12. **minor design rule** - never pass a user-defined type to a function by value.
13. **guideline** - avoid using `short` in the interface; use `int` instead.
14. **guideline** - avoid using `unsigned` in the interface; use `int` instead.

15. **guideline** - explicitly declare (public or private) the constructor and assignment operator for any class defined in a header file.
16. **minor design rule** - in every class that declares or is derived from a class that declares a virtual function, explicitly declare the destructor as the first virtual function in the class and define it out of line.

[^1]: “John Lakos - Large Scale C++ Software Design”.

1.1.3 Development guide

We, at Kagome, enforce **clean architecture** as much as possible.



image

an entity layer

- Entity represents domain object. Example: PeerId, Address, Customer, Socket.
- **MUST** store state (data members). Has interface methods to work with this state.
- **MAY** be represented as plain struct or complex class (developer is free to choose either).
- **MUST NOT** depend on *interfaces* or *implementations*.

an interface layer

- An Interface layer contains C++ classes, which have only pure virtual functions and destructor.
- Classes from the *interface layer* **MAY** depend on the *entity layer*, not vice versa.
- **MUST** have public virtual or protected non-virtual destructor ([cppcoreguidelines](#)).

an implementation layer

- Classes in this layer **MUST** implement one or multiple interfaces from *an interface layer*.
- Example: interface of the map with `put`, `get`, `contains` methods. One class-implementation may use `leveldb` as backend, another class-implementation may use `lmdb` as backend. Then, it is easy to select required implementation at runtime without recompilation. In tests, it is easy to mock the interface.
- **MAY** store state (data members)
- **MUST** have base class-interface with public virtual default destructor

Example:

If it happens that you want to add some functionality to Entity, but functionality depends on some Interface, then create new Interface, which will work with this Entity:

Example:

```
// an Entity
class Block {
public:
    // getters and setters

    // 1. and you want to add hash()
    Buffer hash(); // 2. NEVER DO THIS
private:
    int fieldA;
    string fieldB;
};

// 3. instead, create an interface for "Hasher":
// "Interface" layer
class Hasher {
public:
    virtual ~Hasher() = default;
    virtual Hash256 sha2_256(const Block&b) const = 0;
};

// 4. with implementation. Actual implementation may use openssl, for instance.
// "Implementation" layer
class HasherImpl: public Hasher {
public:
    Hash256 sha2_256(const Block&b) const {
        ... an actual implementation of hash() ...
    }
};
```

Rationale – services can be easily mocked in tests. With the Hasher example, we can create block with predefined hash.

This policy ensures testability and maintainability for projects of any size, from 1KLOC to 10MLOC.

1.1.4 Guide for `outcome::result<T>`

Use `outcome::result<T>` from `<outcome/outcome.hpp>` to represent either value of type `T` or `std::error_code`.

DO NOT DEFINE CUSTOM ERROR TYPES. There is one good explanation for that – one can not merge two custom types automatically, however error codes can be merged.

Please, read <https://ned14.github.io/outcome/> carefully.

Creating `outcome::result<T>`

Example 1 - Enum in namespace:

```
//////////
// mylib.hpp:

#include <outcome/outcome.hpp>

namespace my::super::lib {
    enum class MyError {
        Case1 = 1, // NOTE: MUST NOT start with 0 (it represents success)
        Case2 = 2,
        Case3 = 4, // any codes may be used
        Case4      // or no codes at all
    };

    outcome::result<int> calc(int a, int b);
}
// declare required functions in hpp
// outside of any namespace
OUTCOME_HPP_DECLARE_ERROR(my::super::lib, MyError);

//////////
// mylib.cpp:
#include "mylib.hpp"

// outside of any namespace
OUTCOME_CPP_DEFINE_CATEGORY(my::super::lib, MyError, e){
    using my::super::lib::MyError; // not necessary, just for convenience
    switch(e) {
        case MyError::Case1: return "Case1 message";
        case MyError::Case2: return "Case2 message";
        case MyError::Case3: return "Case3 message";
        case MyError::Case4: return "Case4 message";
        default: return "unknown"; // NOTE: do not forget to handle everything else
    }
}

namespace my::super::lib {
    outcome::result<int> calc(int a, int b){
        // then simply return enum in case of error:
```

(continues on next page)

(continued from previous page)

```

    if(a < 0)    return MyError::Case1;
    if(a > 100)  return MyError::Case2;
    if(b < 0)    return MyError::Case3;
    if(b < 100)  return MyError::Case4;

    return a + b; // simply return value in case of value:
}
}

```

Example 2 - Enum as class member:

```

//////////
// mylib.hpp:

#include <outcome/outcome.hpp>

namespace my::super::lib {

    class MyLib {
    public:
        // MyError now is a member of class
        + enum class MyError {
        +     Case1 = 1, // NOTE: MUST NOT start with 0 (it represents success)
        +     Case2 = 2,
        +     Case3 = 4, // any codes may be used
        +     Case4      // or no codes at all
        + };

        outcome::result<int> calc(int a, int b);
    }
}

// declare required functions in hpp
// outside of any namespace
+// NOTE: 1 args is only namespace, class prefix should be added to enum
-OUTCOME_HPP_DECLARE_ERROR(my::super::lib, MyError);
+OUTCOME_HPP_DECLARE_ERROR(my::super::lib, MyLib::MyError);

//////////
// mylib.cpp:
#include "mylib.hpp"

-OUTCOME_CPP_DEFINE_CATEGORY(my::super::lib, MyError, e){
+OUTCOME_CPP_DEFINE_CATEGORY(my::super::lib, MyLib::MyError, e){
- using my::super::lib::MyError; // not necessary, just for convenience
+ using my::super::lib::MyLib::MyError; // not necessary, just for convenience
switch(e) {
    case MyError::Case1: return "Case1 message";
    case MyError::Case2: return "Case2 message";
    case MyError::Case3: return "Case3 message";
    case MyError::Case4: return "Case4 message";
    default: return "unknown"; // NOTE: do not forget to handle everything else
}
}

```

(continues on next page)

(continued from previous page)

```
namespace my::super::lib {
- outcome::result<int> calc(int a, int b)
+ outcome::result<int> MyLib::calc(int a, int b){
    // then simply return enum in case of error:
    if(a < 0)    return MyError::Case1;
    if(a > 100) return MyError::Case2;
    if(b < 0)    return MyError::Case3;
    if(b > 100) return MyError::Case4;

    return a + b; // simply return value in case of value
}
}
```

Inspecting `outcome::result<T>`

Inspecting is very straightforward:

```
outcome::result<int> calc(int a, int b){
    // then simply return enum in case of error:
    if(a < 0)    return MyError::Case1;
    if(a > 100) return MyError::Case2;
    if(b < 0)    return MyError::Case3;
    if(b < 100) return MyError::Case4;

    return a + b; // simply return value in case of value:
}

outcome::result<int> parent(int a) {
    // NOTE: returns error if calc returned it, otherwise get unwrapped calc result
    OUTCOME_TRY(val, calc(a, 1)); // use convenient macro
    // here val=a+1

    // or

    auto&& result = calc(a, 2);
    if(result) {
        // has value
        auto&& v = result.value();
        return v;
    } else {
        // has error
        auto&& e = result.error(); // get std::error_code
        return e;
    }

    // or

    // pass result to parent
    return calc(a, 3);
}
```


1.1.5 Tooling

In Kagome we use certain set of tools to assure code quality. Here is a list, and guide how to use them.

clang-tidy

Set of rules is specified at root `.clang-tidy` file.

Configure + Build + Run clang-tidy (slow)

1. Ensure clang-tidy is in PATH
2. `mkdir build`
3. `cd build`
4. `cmake .. -DCLANG_TIDY=ON`
5. `make`

Warnings/errors will be reported to stderr, same as compiler warnings/errors.

Run clang-tidy for changes between your branch and master

1. Ensure clang-tidy is in PATH
2. Ensure clang-tidy-diff.py is in PATH.
 - on Mac it is usually located at `/usr/local/Cellar/llvm/8.0.0_1/share/clang/clang-tidy-diff.py` (note, 8.0.0_1 is your version, it may be different).
 - on Linux it is usually located at `/usr/lib/llvm-8/share/clang/clang-tidy-diff.py`
3. `mkdir build`
4. `cd build`
5. `cmake ..`
6. `make generated` - this step creates generated headers (protobuf, etc)
7. `cd ..`
8. `housekeeping/clang-tidy-diff.sh`

Toolchain build

When `CMAKE_TOOLCHAIN_FILE` is specified, then specific toolchain is used. Toolchain is a cmake file, which sets specific variables, such as compiler, its flags, build mode, language standard.

Example:

```
mkdir build
cd build
cmake .. -DCMAKE_TOOLCHAIN_FILE=../cmake/toolchain/gcc-8_cxx17.cmake
```

All dependencies will be built with gcc-8 and cxx17 standard.

Default toolchain is `cxx17`. `cmake`. [List of toolchains](#).

Also, **sanitizers** can be enabled with use of toolchains, so all dependencies will be built with specified sanitizer.

[List of sanitizers available](#).

coverage

Coverage is calculated automatically by Jenkins (CI).

We use [codecov](#) to display coverage.

1.2 CodeStyle

We follow [CppCoreGuidelines](#).

Please use provided `.clang-format` file to autoformat the code.

OVERVIEW

2.1 Getting started

2.1.1 Prerequisites

For now, please refer to the [Dockerfile](#) to get a picture of what you need for a local build-environment.

2.1.2 Clone

```
git clone --recurse-submodules https://github.com/soramitsu/kagome
cd kagome

# Only needed if you did not use `--recurse-submodules` above
git submodule update --init --recursive
```

2.1.3 Build

First build will likely take long time. However, you can cache binaries to [hunter-binary-cache](#) or even download binaries from the cache in case someone has already compiled project with the same compiler. To this end, you need to set up two environment variables:

```
GITHUB_HUNTER_USERNAME=<github account name>
GITHUB_HUNTER_TOKEN=<github token>
```

To generate github token follow the [instructions](#). Make sure `read:packages` and `write:packages` permissions are granted (step 7 in instructions).

Build all

This project is can be built with

```
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j
```

Tests can be run with:

```
cd build
ctest
```

Build node application

If you'd like to build node use the following instruction

```
mkdir build && cd build  
cmake -DCMAKE_BUILD_TYPE=Release ..  
make kagome -j
```

TUTORIALS

In this folder we have our awesome Kagome Tutorials. Choose the one you'd like to walk through below:

3.1 Your first Kagome chain

In this tutorial you will learn how to execute Kagome-based Polkadot-host chain which can be used as a cryptocurrency, and interact with it by sending extrinsics and executing queries.

3.1.1 Prerequisites

1. Kagome validating node binary built as described [here](#).
2. For your convenience make sure you have this binary included into your path:

```
# from Kagome's root repo:  
PATH=$PATH:$ (pwd) /build/node/
```

3. Python 3 installed in the system with `substrate-interface` package installed

If you are not sure if you have `requests` package installed in your python run `pip3 install substrate-interface`

3.1.2 Tutorial

During this tutorial we will:

1. Launch Kagome network using prepared genesis file
2. Query the balance of Alice account
3. Send extrinsic that transfers some amount of currency from Alice to Bob
4. Query again the balance of Alice to make sure balance was updated

Launch Kagome network

For this tutorial we will spin up a simple network of a single peer with predefined genesis.

To start with let's navigate into the node's folder:

```
cd examples/first_kagome_chain
```

`first_kagome_chain` folder contains necessary configuration files for our tutorial:

- `localchain.json` – genesis file for our network. It contains necessary key-value pairs that should be inserted before the genesis block
- `base_path` – Directory, containing kagome base path. It contains several dirs, each one named with the chain id, which data it stores (`dev` in this case). Data for each chain consists of `db/` (will be initialized on node startup) and `keystore/` (keys to sign the messages sent by our authority). The latter has to exist prior to the node start. This behaviour will be improved in the future.

`localchain.json` contains Alice and Bob accounts. Both have 999998900.0 amount of crypto. Their keys can be generated using `subkey` tool:

```
subkey inspect //Alice
# Secret Key URI `//Alice` is account:
# Secret seed:      0xe5be9a5092b81bca64be81d212e7f2f9eba183bb7a90954f7b76361f6edb5c0a
# Public key (hex): 0xd43593c715fdd31c61141abd04a99fd6822c8558854ccde39a5684e7a56da27d
# Account ID:      0xd43593c715fdd31c61141abd04a99fd6822c8558854ccde39a5684e7a56da27d
# SS58 Address:    5GrwvaEF5zXb26Fz9rcQpDWS57CtERHPNehXCPcNoHGKutQY

subkey inspect //Bob
# Secret Key URI `//Bob` is account:
# Secret seed:      0x398f0c28f98885e046333d4a41c19cee4c37368a9832c6502f6cfd182e2aef89
# Public key (hex): 0x8eaf04151687736326c9fea17e25fc5287613693c912909cb226aa4794f26a48
# Account ID:      0x8eaf04151687736326c9fea17e25fc5287613693c912909cb226aa4794f26a48
# SS58 Address:    5FHneW46xGXgs5mUiveU4sbTyGBzmstUspZC92UhjJM694ty
```

If you are running this tutorial not for the first time, then make sure you cleaned up your storage as follows (assuming storage files are generated in `ldb/` folder):

```
rm -rf ldb
```

For this tutorial you can start a single node network as follows:

```
kagome \
  --validator \
  --chain localchain.json \
  --base-path base_path \
  --port 30363 \
  --rpc-port 9933 \
  --ws-port 9944
```

Let's look at this flags in detail:

More flags info available by running `kagome --help`.

You should see the log messages notifying about produced and finalized the blocks.

Now chain is running on a single node. To query it we can use `localhost`'s ports 9933 for http- and 9944 for websockets-based RPCs.

Kagome blockchain is constantly producing new blocks, even if there were no transactions in the network.

Query the balance

Now open second terminal and go to the transfer folder, available from the projects root directory.

```
cd examples/transfer
```

This folder contains two python scripts:

1. `balance.py <address> <account_id>` – executes query to kagome, which returns balance of provided account
 - `<address>` address node's http service
 - `<account_id>` id of account being queried
2. `transfer.py <address> <seed> <dest> <amount>` – sends provided extrinsic
 - `<address>` address node's http service
 - `<seed>` secret seed of source account
 - `<dest>` destination account
 - `<amount>` amount of crypto to be transferred

Let's query current balance of Alice's account.

```
python3 balance.py localhost:9933 5GrwvaEF5zXb26Fz9rcQpDWS57CtERHpNehXCPcNoHGKutQY
# Current free balance: 10000.0
```

Let's do the same for the Bob's account.

```
python3 balance.py localhost:9933 5FHneW46xGXgs5mUiveU4sbTyGBzmstUspZC92UhjJM694ty
# Current free balance: 10000.0
```

Send extrinsic

We can generate extrinsic using the following command:

This command will create extrinsic that transfers 1 from Alice to Bob's account (Alice's account is defined by secret seed and Bob's account by account id).

To send extrinsic use `transfer.py` script as follows:

```
python3 transfer.py localhost:9933 \
  ↪0xe5be9a5092b81bca64be81d212e7f2f9eba183bb7a90954f7b76361f6edb5c0a \
  ↪5FHneW46xGXgs5mUiveU4sbTyGBzmstUspZC92UhjJM694ty 1
Extrinsic '0x315060176633a3e20656bd15c6ecef63b9f8bdc45595dc8ad52a02ae38d1708' sent
```

Query again the balances

Now let's check that extrinsic was actually applied:

Get the balance of Bob's account:

```
python3 balance.py localhost:9933 5FHneW46xGXgs5mUiveU4sbTyGBzmstUspZC92UhjJM694ty
# Current free balance: 10001.0
```

We can see that Bob's balance was increased by 1 as it was set on the subkey command

Now let's check Alice's account:

```
python3 balance.py localhost:9933 5GrwvaEF5zXb26Fz9rcQpDWS57CtERHpNehXCPcNoHGKutQY
Current free balance: 9998.999829217584
```

We can see that Alice's account was decreased by more than 1. This is caused by the commission paid for transfer

3.1.3 Conclusion

Now you know how to set up single peer kagome network and execute transactions on it

3.2 Start private Kagome network

In this tutorial we will learn how to start a blockchain network with a validator and block producing nodes

First go to tutorial's folder:

```
cd examples/network
```

3.2.1 Execute first validating node

First we execute validating node in the similar way we did it during previous tutorial. This node will produce and finalize blocks.

To start with let's navigate into the node's folder:

```
kagome \
  --validator \
  --chain testchain.json \
  --base-path validating1 \
  --port 11122 \
  --rpc-port 11133 \
  --ws-port 11144
```

3.2.2 Execute second validating node (node with authority)

Now that validating node is up and running, second node can join the network by bootstrapping from the first node. Command will look very similar.

```
kagome \
  --validator \
  --chain testchain.json \
  --base-path validating2 \
  --port 11222 \
  --rpc-port 11233 \
  --ws-port 11244
```

Second node passes several steps before actual block production begins:

1. Waiting for block announcements to understand which blocks are missing.
2. Synchronize blocks between the latest synchronized one and the received one
3. Listen for several blocks, to figure out the slot time

4. Start block production when slot time is calculated using [median algorithm](#)

Because these two nodes are running on the same machine, second node must be specified with different port numbers

Note that both nodes have the same hash of block 0: 2b32173d63796278d1cea23fcb255866153f07700226f3d7ba348e25

3.2.3 Execute syncing (without authority) node

Syncing node cannot participate in either block production or block finalization. However, it can connect to the network and import all produced blocks. Besides that, syncing node can also receive extrinsics and broadcast them to the network.

To start syncing node kagome binary is used as follows:

```
kagome \
  --chain testchain.json \
  --base-path syncing1 \
  --port 21122 \
  --rpc-port 21133 \
  --ws-port 21144
```

Note that trie root is the same with validating nodes. When syncing node receives block announcement it first synchronizes missing blocks and then listens to the new blocks and finalization.

3.2.4 Send transaction

Like in previous tutorial we will send transfer from Alice to Bob to check that transaction was applied on every node.

We can send transaction on any of the node, as it will be propagated to the block producing nodes and stored in their transaction pools until transactions is included to the block:

```
# from kagome root directory
cd examples/transfer
python3 transfer.py localhost:9933 \
  ↪ 0xe5be9a5092b81bca64be81d212e7f2f9eba183bb7a90954f7b76361f6edb5c0a \
  ↪ 5FHneW46xGXgs5mUiveU4sbTyGBzmstUspZC92UhjJM694ty 2
```


KAGOME IN MEDIA

- Press-release: [Soramitsu to implement Polkadot Runtime Environment in C++](#)
- [Kagome: C++ implementation of PRE](#) presentation at DOTCon (18.08.19)
- [Kagome and consensus in Polkadot](#) presentation (in Russian) during Innopolis blockchain meetup (28.10.19)