The Move Book

In this chapter, you will learn how to create a new package, write a simple module, compile it, and run tests with the Move CLI. Make sure you have installed Sui and set up your IDE environment . Run the command below to test if Sui has been installed correctly.

Move CLI is a command-line interface for the Move language; it is built into the Sui binary and provides a set of commands to manage packages, compile and test code.

The structure of the chapter is as follows:

To create a new program, we will use the sui move new command followed by the name of the application. Our first program will be called hello_world .

Note: In this and other chapters, if you see code blocks with lines starting with $ (dollar sign), it means that the following command should be run in a terminal. The sign should not be included. It's a common way of showing commands in terminal environments.

The sui move command gives access to the Move CLI - a built-in compiler, test runner and a utility for all things Move. The new command followed by the name of the package will create a new package in a new folder. In our case, the folder name is "hello_world".

We can view the contents of the folder to see that the package was created successfully.

Move CLI will create a scaffold of the application and pre-create the directory structure and all necessary files. Let's see what's inside.

The Move.toml file, known as the package manifest , contains definitions and configuration settings for the package. It is used by the Move Compiler to manage package metadata, fetch dependencies, and register named addresses. We will explain it in detail in the Concepts chapter.

By default, the package features one named address - the name of the package.

The sources/ directory contains the source files. Move source files have .move extension, and are typically named after the module defined in the file. For example, in our case, the file name is hello_world.move and the Move CLI has already placed commented out code inside:

The / *and* / are the comment delimiters in Move. Everything in between is ignored by the compiler and can be used for documentation or notes. We explain all ways to comment the code in the Basic Syntax .

The commented out code is a module definition, it starts with the keyword module followed by a named address (or an address literal), and the module name. The module name is a unique identifier for the module and has to be unique within the package. The module name is used to reference the module from other modules or transactions.

The tests/ directory contains package tests. The compiler excludes these files in the regular build process but uses them in test and dev modes. The tests are written in Move and are marked with the #[test] attribute. Tests can be grouped in a separate module (then it's usually called module_name_tests.move), or inside the module they're testing.

Modules, imports, constants and functions can be annotated with #[test_only] . This attribute is used to exclude modules, functions or imports from the build process. This is useful when you want to add helpers for your tests without including them in the code that will be published on chain.

The hello_world_tests.move file contains a commented out test module template:

Additionally, Move CLI supports the examples/ folder. The files there are treated similarly to the ones placed under the tests/ folder - they're only built in the test and dev modes. They are to be examples of how to use the package or how to integrate it with other packages. The most popular use case is for documentation purposes and library packages.

Move is a compiled language, and as such, it requires the compilation of source files into Move Bytecode. It contains only necessary information about the module, its members, and types, and excludes comments and some identifiers (for example, for constants).

To demonstrate these features, let's replace the contents of the sources/hello_world.move file with the following:

During compilation, the code is built, but not run. A compiled package only includes functions that can be called by other modules or in a transaction. We will explain these concepts in the Concepts chapter. But now, let's see what happens when we run the sui move

build .

It should output the following message on your console.

During the compilation, Move Compiler automatically creates a build folder where it places all fetched and compiled dependencies as well as the bytecode for the modules of the current package.

If you're using a versioning system, such as Git, build folder should be ignored. For example, you should use a .gitignore file and add build to it.

Before we get to testing, we should add a test. Move Compiler supports tests written in Move and provides the execution environment. The tests can be placed in both the source files and in the tests/ folder. Tests are marked with the #[test] attribute and are automatically discovered by the compiler. We explain tests in depth in the Testing section.

Replace the contents of the tests/hello_world_tests.move with the following content:

Here we import the hello_world module, and call its hello_world function to test that the output is indeed the string "Hello, World!". Now, that we have tests in place, let's compile the package in the test mode and run tests. Move CLI has the test command for this:

The output should be similar to the following:

If you're running the tests outside of the package folder, you can specify the path to the package:

You can also run a single or multiple tests at once by specifying a string. All the tests names containing the string will be run:

In this section, we explained the basics of a Move package: its structure, the manifest, the build, and test flows. On the next page , we will write an application and see how the code is structured and what the language can do.

Create a New Package

To create a new program, we will use the sui move new command followed by the name of the application. Our first program will be called hello_world .

Note: In this and other chapters, if you see code blocks with lines starting with $ (dollar sign), it means that the following command should be run in a terminal. The sign should not be included. It's a common way of showing commands in terminal environments.

```
bash $ sui move new hello_world
```

The sui move command gives access to the Move CLI - a built-in compiler, test runner and a utility for all things Move. The new command followed by the name of the package will create a new package in a new folder. In our case, the folder name is "hello_world".

We can view the contents of the folder to see that the package was created successfully.

```
bash $ ls -l hello_world Move.toml sources tests
```

Move CLI will create a scaffold of the application and pre-create the directory structure and all necessary files. Let's see what's inside.

```
bash hello_world ├── Move.toml ├── sources │ └── hello_world.move └── tests └──
hello_world_tests.move
```

The Move.toml file, known as the package manifest , contains definitions and configuration settings for the package. It is used by the Move Compiler to manage package metadata, fetch dependencies, and register named addresses. We will explain it in detail in the Concepts chapter.

By default, the package features one named address - the name of the package.

```
bash [addresses] hello_world = "0x0"
```

The sources/ directory contains the source files. Move source files have .move extension, and are typically named after the module defined in the file. For example, in our case, the file name is hello_world.move and the Move CLI has already placed commented out code inside:

```
bash /* /// Module: hello_world module hello_world::hello_world; */
```

The / *and* / are the comment delimiters in Move. Everything in between is ignored by the compiler and can be used for documentation or notes. We explain all ways to comment the code in the Basic Syntax .

The commented out code is a module definition, it starts with the keyword module followed by a named address (or an address literal), and the module name. The module name is a unique identifier for the module and has to be unique within the package. The module name is used to reference the module from other modules or transactions.

The tests/ directory contains package tests. The compiler excludes these files in the regular build process but uses them in test and dev modes. The tests are written in Move and are marked with the #[test] attribute. Tests can be grouped in a separate module (then it's usually called module_name_tests.move), or inside the module they're testing.

Modules, imports, constants and functions can be annotated with #[test_only] . This attribute is used to exclude modules, functions or imports from the build process. This is useful when you want to add helpers for your tests without including them in the code that will be published on chain.

The hello_world_tests.move file contains a commented out test module template:

```bash /*

# [test_only]

module hello_world::hello_world_tests; // uncomment this line to import the module // use hello_world::hello_world;

const ENotImplemented: u64 = 0;

# [test]

fun test_hello_world() { // pass }

# [test, expected_failure(abort_code = hello_world::hello_world_tests::ENotImplemented)]

fun test_hello_world_fail() { abort ENotImplemented } */ ```

Additionally, Move CLI supports the examples/ folder. The files there are treated similarly to the ones placed under the tests/ folder - they're only built in the test and dev modes. They are to be examples of how to use the package or how to integrate it with other packages. The most popular use case is for documentation purposes and library packages.

Move is a compiled language, and as such, it requires the compilation of source files into Move Bytecode. It contains only necessary information about the module, its members, and types, and excludes comments and some identifiers (for example, for constants).

To demonstrate these features, let's replace the contents of the sources/hello_world.move file with the following:

``bash /// The modulehello_worldunder named addresshello_world. /// The named address is set in theMove.toml`. module hello_world::hello_world;

// Imports the `String` type from the Standard Library use std::string::String;

/// Returns the "Hello, World!" as a `String`. public fun hello_world(): String { b"Hello, World!".to_string() } ```

During compilation, the code is built, but not run. A compiled package only includes functions that can be called by other modules or in a transaction. We will explain these concepts in the Concepts chapter. But now, let's see what happens when we run the sui move build .

```bash

# run from the `hello_world` folder

$ sui move build

# alternatively, if you didn't `cd` into it

$ sui move build --path hello_world ```

It should output the following message on your console.

```
bash UPDATING GIT DEPENDENCY https://github.com/MystenLabs/sui.git INCLUDING DEPENDENCY Bridge
INCLUDING DEPENDENCY DeepBook INCLUDING DEPENDENCY SuiSystem INCLUDING DEPENDENCY Sui INCLUDING
DEPENDENCY MoveStdlib BUILDING hello_world
```

During the compilation, Move Compiler automatically creates a build folder where it places all fetched and compiled dependencies as well as the bytecode for the modules of the current package.

If you're using a versioning system, such as Git, build folder should be ignored. For example, you should use a .gitignore file and add build to it.

Before we get to testing, we should add a test. Move Compiler supports tests written in Move and provides the execution environment. The tests can be placed in both the source files and in the tests/ folder. Tests are marked with the #[test] attribute and are automatically discovered by the compiler. We explain tests in depth in the Testing section.

Replace the contents of the tests/hello_world_tests.move with the following content:

```bash

# [test_only]

module hello_world::hello_world_tests;

use hello_world::hello_world;

# [test]

fun test_hello_world() { assert!(hello_world::hello_world() == b"Hello, World!".to_string(), 0); } ```

Here we import the hello_world module, and call its hello_world function to test that the output is indeed the string "Hello, World!". Now, that we have tests in place, let's compile the package in the test mode and run tests. Move CLI has the test command for this:

```
bash $ sui move test
```

The output should be similar to the following:

```
bash INCLUDING DEPENDENCY Bridge INCLUDING DEPENDENCY DeepBook INCLUDING DEPENDENCY SuiSystem
INCLUDING DEPENDENCY Sui INCLUDING DEPENDENCY MoveStdlib BUILDING hello_world Running Move unit
tests [ PASS ] 0x0::hello_world_tests::test_hello_world Test result: OK. Total tests: 1; passed: 1;
failed: 0
```

If you're running the tests outside of the package folder, you can specify the path to the package:

```
bash $ sui move test --path hello_world
```

You can also run a single or multiple tests at once by specifying a string. All the tests names containing the string will be run:

```
bash $ sui move test test_hello
```

In this section, we explained the basics of a Move package: its structure, the manifest, the build, and test flows. On the next page , we will write an application and see how the code is structured and what the language can do.

Directory Structure

Move CLI will create a scaffold of the application and pre-create the directory structure and all necessary files. Let's see what's inside.

```
bash hello_world ├── Move.toml ├── sources │ └── hello_world.move └── tests └──
hello_world_tests.move
```
```

The Move.toml file, known as the package manifest , contains definitions and configuration settings for the package. It is used by the Move Compiler to manage package metadata, fetch dependencies, and register named addresses. We will explain it in detail in the Concepts chapter.

By default, the package features one named address - the name of the package.

```
bash [addresses] hello_world = "0x0"
```

The sources/ directory contains the source files. Move source files have .move extension, and are typically named after the module defined in the file. For example, in our case, the file name is hello_world.move and the Move CLI has already placed commented out code inside:

```
bash /* /// Module: hello_world module hello_world::hello_world; */
```

The / *and* / are the comment delimiters in Move. Everything in between is ignored by the compiler and can be used for documentation or notes. We explain all ways to comment the code in the Basic Syntax .

The commented out code is a module definition, it starts with the keyword module followed by a named address (or an address literal), and the module name. The module name is a unique identifier for the module and has to be unique within the package. The module name is used to reference the module from other modules or transactions.

The tests/ directory contains package tests. The compiler excludes these files in the regular build process but uses them in test and dev modes. The tests are written in Move and are marked with the #[test] attribute. Tests can be grouped in a separate module (then it's usually called module_name_tests.move), or inside the module they're testing.

Modules, imports, constants and functions can be annotated with #[test_only] . This attribute is used to exclude modules, functions or imports from the build process. This is useful when you want to add helpers for your tests without including them in the code that will be published on chain.

The hello_world_tests.move file contains a commented out test module template:

```bash /*

# [test_only]

module hello_world::hello_world_tests; // uncomment this line to import the module // use hello_world::hello_world;

const ENotImplemented: u64 = 0;

# [test]

fun test_hello_world() { // pass }

# [test, expected_failure(abort_code = hello_world::hello_world_tests::ENotImplemented)]

fun test_hello_world_fail() { abort ENotImplemented } */ ```
```

Additionally, Move CLI supports the examples/ folder. The files there are treated similarly to the ones placed under the tests/ folder - they're only built in the test and dev modes. They are to be examples of how to use the package or how to integrate it with other packages. The most popular use case is for documentation purposes and library packages.

Move is a compiled language, and as such, it requires the compilation of source files into Move Bytecode. It contains only necessary information about the module, its members, and types, and excludes comments and some identifiers (for example, for constants).

To demonstrate these features, let's replace the contents of the sources/hello_world.move file with the following:

```bash /// The modulehello_worldunder named addresshello_world. /// The named address is set in theMove.toml`. module hello_world::hello_world;

// Imports the String type from the Standard Library use std::string::String;
```

/// Returns the "Hello, World!" as a `String`. public fun hello_world(): String { b"Hello, World!".to_string() } ```

During compilation, the code is built, but not run. A compiled package only includes functions that can be called by other modules or in a transaction. We will explain these concepts in the Concepts chapter. But now, let's see what happens when we run the sui move build .

```bash

# run from the `hello_world` folder

$ sui move build

# alternatively, if you didn't `cd` into it

$ sui move build --path hello_world ```

It should output the following message on your console.

```
bash UPDATING GIT DEPENDENCY https://github.com/MystenLabs/sui.git INCLUDING DEPENDENCY Bridge
INCLUDING DEPENDENCY DeepBook INCLUDING DEPENDENCY SuiSystem INCLUDING DEPENDENCY Sui INCLUDING
DEPENDENCY MoveStdlib BUILDING hello_world
```

During the compilation, Move Compiler automatically creates a build folder where it places all fetched and compiled dependencies as well as the bytecode for the modules of the current package.

If you're using a versioning system, such as Git, build folder should be ignored. For example, you should use a .gitignore file and add build to it.

Before we get to testing, we should add a test. Move Compiler supports tests written in Move and provides the execution environment. The tests can be placed in both the source files and in the tests/ folder. Tests are marked with the #[test] attribute and are automatically discovered by the compiler. We explain tests in depth in the Testing section.

Replace the contents of the tests/hello_world_tests.move with the following content:

```bash

# [test_only]

module hello_world::hello_world_tests;

use hello_world::hello_world;

# [test]

fun test_hello_world() { assert!(hello_world::hello_world() == b"Hello, World!".to_string(), 0); } ```

Here we import the hello_world module, and call its hello_world function to test that the output is indeed the string "Hello, World!". Now, that we have tests in place, let's compile the package in the test mode and run tests. Move CLI has the test command for this:

```
bash $ sui move test
```

The output should be similar to the following:

```
bash INCLUDING DEPENDENCY Bridge INCLUDING DEPENDENCY DeepBook INCLUDING DEPENDENCY SuiSystem
INCLUDING DEPENDENCY Sui INCLUDING DEPENDENCY MoveStdlib BUILDING hello_world Running Move unit
tests [ PASS ] 0x0::hello_world_tests::test_hello_world Test result: OK. Total tests: 1; passed: 1;
failed: 0
```

If you're running the tests outside of the package folder, you can specify the path to the package:

```
bash $ sui move test --path hello_world
```

You can also run a single or multiple tests at once by specifying a string. All the tests names containing the string will be run:

```
bash $ sui move test test_hello
```

In this section, we explained the basics of a Move package: its structure, the manifest, the build, and test flows. On the next page , we will write an application and see how the code is structured and what the language can do.

Compiling the Package

Move is a compiled language, and as such, it requires the compilation of source files into Move Bytecode. It contains only necessary information about the module, its members, and types, and excludes comments and some identifiers (for example, for constants).

To demonstrate these features, let's replace the contents of the sources/hello_world.move file with the following:

``bash /// The modulehello_worldunder named addresshello_world. /// The named address is set in theMove.toml`. module hello_world::hello_world;

// Imports the `String` type from the Standard Library use std::string::String;

/// Returns the "Hello, World!" as a `String`. public fun hello_world(): String { b"Hello, World!".to_string() } ```

During compilation, the code is built, but not run. A compiled package only includes functions that can be called by other modules or in a transaction. We will explain these concepts in the Concepts chapter. But now, let's see what happens when we run the sui move build .

```bash

# run from the `hello_world` folder

$ sui move build

# alternatively, if you didn't `cd` into it

$ sui move build --path hello_world ```

It should output the following message on your console.

```
bash UPDATING GIT DEPENDENCY https://github.com/MystenLabs/sui.git INCLUDING DEPENDENCY Bridge
INCLUDING DEPENDENCY DeepBook INCLUDING DEPENDENCY SuiSystem INCLUDING DEPENDENCY Sui INCLUDING
DEPENDENCY MoveStdlib BUILDING hello_world
```

During the compilation, Move Compiler automatically creates a build folder where it places all fetched and compiled dependencies as well as the bytecode for the modules of the current package.

If you're using a versioning system, such as Git, build folder should be ignored. For example, you should use a .gitignore file and add build to it.

Before we get to testing, we should add a test. Move Compiler supports tests written in Move and provides the execution environment. The tests can be placed in both the source files and in the tests/ folder. Tests are marked with the #[test] attribute and are automatically discovered by the compiler. We explain tests in depth in the Testing section.

Replace the contents of the tests/hello_world_tests.move with the following content:

```bash

# [test_only]

module hello_world::hello_world_tests;

use hello_world::hello_world;

# [test]

fun test_hello_world() { assert!(hello_world::hello_world() == b"Hello, World!".to_string(), 0); } ```

Here we import the hello_world module, and call its hello_world function to test that the output is indeed the string "Hello, World!". Now, that we have tests in place, let's compile the package in the test mode and run tests. Move CLI has the test command for this:

```
bash $ sui move test
```

The output should be similar to the following:

```
bash INCLUDING DEPENDENCY Bridge INCLUDING DEPENDENCY DeepBook INCLUDING DEPENDENCY SuiSystem
INCLUDING DEPENDENCY Sui INCLUDING DEPENDENCY MoveStdlib BUILDING hello_world Running Move unit
tests [ PASS ] 0x0::hello_world_tests::test_hello_world Test result: OK. Total tests: 1; passed: 1;
failed: 0
```

If you're running the tests outside of the package folder, you can specify the path to the package:

```
bash $ sui move test --path hello_world
```

You can also run a single or multiple tests at once by specifying a string. All the tests names containing the string will be run:

```
bash $ sui move test test_hello
```

In this section, we explained the basics of a Move package: its structure, the manifest, the build, and test flows. On the next page , we will write an application and see how the code is structured and what the language can do.

Running Tests

Before we get to testing, we should add a test. Move Compiler supports tests written in Move and provides the execution environment. The tests can be placed in both the source files and in the tests/ folder. Tests are marked with the #[test] attribute and are automatically discovered by the compiler. We explain tests in depth in the Testing section.

Replace the contents of the tests/hello_world_tests.move with the following content:

```bash

# [test_only]

module hello_world::hello_world_tests;

use hello_world::hello_world;

# [test]

fun test_hello_world() { assert!(hello_world::hello_world() == b"Hello, World!".to_string(), 0); } ```

Here we import the hello_world module, and call its hello_world function to test that the output is indeed the string "Hello, World!". Now, that we have tests in place, let's compile the package in the test mode and run tests. Move CLI has the test command for this:

```
bash $ sui move test
```

The output should be similar to the following:

```
bash INCLUDING DEPENDENCY Bridge INCLUDING DEPENDENCY DeepBook INCLUDING DEPENDENCY SuiSystem
INCLUDING DEPENDENCY Sui INCLUDING DEPENDENCY MoveStdlib BUILDING hello_world Running Move unit
tests [ PASS ] 0x0::hello_world_tests::test_hello_world Test result: OK. Total tests: 1; passed: 1;
failed: 0
```

If you're running the tests outside of the package folder, you can specify the path to the package:

```
bash $ sui move test --path hello_world
```

You can also run a single or multiple tests at once by specifying a string. All the tests names containing the string will be run:
```

```
bash $ sui move test test_hello
```

In this section, we explained the basics of a Move package: its structure, the manifest, the build, and test flows. On the next page , we will write an application and see how the code is structured and what the language can do.

Next Steps

In this section, we explained the basics of a Move package: its structure, the manifest, the build, and test flows. On the next page , we will write an application and see how the code is structured and what the language can do.

Further Reading