Build and Test Packages

If you followed Write a Move Package , you have a basic module that you need to build. If you didn't, then either start with that topic or use your package, substituting that information where appropriate.

Make sure your terminal or console is in the directory that contains your package (my_first_package if you're following along). Use the following command to build your package:

A successful build returns a response similar to the following:

If the build fails, you can use the verbose error messaging in output to troubleshoot and resolve root issues.

Now that you have designed your asset and its accessor functions, it's time to test the package code before publishing.

Sui includes support for the Move testing framework. Using the framework, you can write unit tests that analyze Move code much like test frameworks for other languages, such as the built-in Rust testing framework or the JUnit framework for Java.

An individual Move unit test is encapsulated in a public function that has no parameters, no return values, and has the #[test] annotation. The testing framework executes such functions when you call the sui move test command from the package root (my_move_package directory as per the current running example):

If you execute this command for the package created in Write a Package , you see the following output. Unsurprisingly, the test result has an OK status because there are no tests written yet to fail.

To actually test your code, you need to add test functions. Start with adding a basic test function to the example.move file, inside the module definition:

As the code shows, the unit test function (test_sword_create()) creates a dummy instance of the TxContext struct and assigns it to ctx . The function then creates a sword object using ctx to create a unique identifier and assigns 42 to the magic parameter and 7 to strength . Finally, the test calls the magic and strength accessor functions to verify that they return correct values.

The function passes the dummy context, ctx , to the object::new function as a mutable reference argument (&mut), but passes sword to its accessor functions as a read-only reference argument, &sword .

Now that you have a test function, run the test command again:

After running the test command, however, you get a compilation error instead of a test result:

The error message contains all the necessary information to debug the code. The faulty code is meant to highlight one of the Move language's safety features.

The Sword struct represents a game asset that digitally mimics a real-world item. Obviously, a real sword cannot simply disappear (though it can be explicitly destroyed), but there is no such restriction on a digital one. In fact, this is exactly what's happening in the test function - you create an instance of a Sword struct that simply disappears at the end of the function call. If you saw something disappear before your eyes, you'd be dumbfounded, too.

One of the solutions (as suggested in the error message), is to add the drop ability to the definition of the Sword struct, which would allow instances of this struct to disappear (be dropped). The ability to drop a valuable asset is not a desirable asset property in this case, so another solution is needed. Another way to solve this problem is to transfer ownership of the sword .

To get the test to work, we will need to use the transfer module, which is imported by default. Add the following lines to the end of the test function (after the assert! call) to transfer ownership of the sword to a freshly created dummy address:

Run the test command again. Now the output shows a single successful test has run:

Use a filter string to run only a matching subset of the unit tests. With a filter string provided, the sui move test checks the fully qualified (

::::) name for a match.

Example:

The previous command runs all tests whose name contains sword .

You can discover more testing options through:

The previous testing example uses Move but isn't specific to Sui beyond using some Sui packages, such as sui::tx_context and sui::transfer . While this style of testing is already useful for writing Move code for Sui, you might also want to test additional Sui-specific features. In particular, a Move call in Sui is encapsulated in a Sui transaction, and you might want to test interactions between different transactions within a single test (for example, one transaction creating an object and the other one transferring it).

Sui-specific testing is supported through the test_scenario module that provides Sui-related testing functionality otherwise unavailable in pure Move and its testing framework.

The test_scenario module provides a scenario that emulates a series of Sui transactions, each with a potentially different user executing them. A test using this module typically starts the first transaction using the test_scenario::begin function. This function takes an address of the user executing the transaction as its argument and returns an instance of the Scenario struct representing a scenario.

An instance of the Scenario struct contains a per-address object pool emulating Sui object storage, with helper functions provided to manipulate objects in the pool. After the first transaction finishes, subsequent test transactions start with the test_scenario::next_tx function. This function takes an instance of the Scenario struct representing the current scenario and an address of a user as arguments.

Update your example.move file to include a function callable from Sui that implements sword creation. With this in place, you can then add a multi-transaction test that uses the test_scenario module to test these new capabilities. Put this functions after the accessors (Part 5 in comments).

The code of the new functions uses struct creation and Sui-internal modules (tx_context) in a way similar to what you have seen in the previous sections. The important part is for the function to have correct signatures.

With the new function included, add another test function to make sure it behaves as expected.

There are some details of the new testing function to pay attention to. The first thing the code does is create some addresses that represent users participating in the testing scenario. The test then creates a scenario by starting the first transaction on behalf of the initial sword owner.

The initial owner then executes the second transaction (passed as an argument to the test_scenario::next_tx function), who then transfers the sword they now own to the final owner. In pure Move there is no notion of Sui storage; consequently, there is no easy way for the emulated Sui transaction to retrieve it from storage. This is where the test_scenario module helps - its take_from_sender function allows an address-owned object of a given type (Sword) executing the current transaction to be available for Move code manipulation. For now, assume that there is only one such object. In this case, the test transfers the object it retrieves from storage to another address.

Transaction effects, such as object creation and transfer become visible only after a given transaction completes. For example, if the second transaction in the running example created a sword and transferred it to the administrator's address, it would only become available for retrieval from the administrator's address (via test_scenario , take_from_sender , or take_from_address functions) in the third transaction.

The final owner executes the third and final transaction that retrieves the sword object from storage and checks if it has the expected properties. Remember, as described in Testing a package , in the pure Move testing scenario, after an object is available in Move code (after creation or retrieval from emulated storage), it cannot simply disappear.

In the pure Move testing function, the function transfers the sword object to the fake address to handle the disappearing problem. The test_scenario package provides a more elegant solution, however, which is closer to what happens when Move code actually executes in the context of Sui - the package simply returns the sword to the object pool using the test_scenario::return_to_sender function. For scenarios where returning to the sender is not desirable or if you would like to simply destroy the object, the test_utils module also provides the generic destroy function, that can be used on any type T regardless of its ability. It is advisable to check out other useful functions in the test_scenario and test_utils modules as well.

Run the test command again to see two successful tests for our module:

Each module in a package can include a special initializer function that runs at publication time. The goal of an initializer function is to pre-initialize module-specific data (for example, to create singleton objects). The initializer function must have the following properties for it to execute at publication:

For example, the following init functions are all valid:

While the sui move command does not support publishing explicitly, you can still test module initializers using the testing framework by dedicating the first transaction to executing the initializer function.

The init function for the module in the running example creates a Forge object.

The tests you have so far call the init function, but the initializer function itself isn't tested to ensure it properly creates a Forge object. To test this functionality, add a new_sword function to take the forge as a parameter and to update the number of created swords at the end of the function. If this were an actual module, you'd replace the sword_create function with new_sword . To keep the existing tests from failing, however, we will keep both functions.

Now, create a function to test the module initialization:

As the new test function shows, the first transaction (explicitly) calls the initializer. The next transaction checks if the Forge object has been created and properly initialized. Finally, the admin uses the Forge to create a sword and transfer it to the initial owner.

You can refer to the source code for the package (with all the tests and functions properly adjusted) in the first_package module in the sui/examples directory. You can also use the following toggle to review the complete code.

example.move

Building your package

Make sure your terminal or console is in the directory that contains your package (my_first_package if you're following along). Use the following command to build your package:

A successful build returns a response similar to the following:

If the build fails, you can use the verbose error messaging in output to troubleshoot and resolve root issues.

Now that you have designed your asset and its accessor functions, it's time to test the package code before publishing.

Sui includes support for the Move testing framework. Using the framework, you can write unit tests that analyze Move code much like test frameworks for other languages, such as the built-in Rust testing framework or the JUnit framework for Java.

An individual Move unit test is encapsulated in a public function that has no parameters, no return values, and has the #[test] annotation. The testing framework executes such functions when you call the sui move test command from the package root (my_move_package directory as per the current running example):

If you execute this command for the package created in Write a Package , you see the following output. Unsurprisingly, the test result has an OK status because there are no tests written yet to fail.

To actually test your code, you need to add test functions. Start with adding a basic test function to the example.move file, inside the module definition:

As the code shows, the unit test function (test_sword_create()) creates a dummy instance of the TxContext struct and assigns it to ctx . The function then creates a sword object using ctx to create a unique identifier and assigns 42 to the magic parameter and 7 to strength . Finally, the test calls the magic and strength accessor functions to verify that they return correct values.

The function passes the dummy context, ctx , to the object::new function as a mutable reference argument (&mut), but passes sword to its accessor functions as a read-only reference argument, &sword .

Now that you have a test function, run the test command again:

After running the test command, however, you get a compilation error instead of a test result:

The error message contains all the necessary information to debug the code. The faulty code is meant to highlight one of the Move language's safety features.

The Sword struct represents a game asset that digitally mimics a real-world item. Obviously, a real sword cannot simply

disappear (though it can be explicitly destroyed), but there is no such restriction on a digital one. In fact, this is exactly what's happening in the test function - you create an instance of a Sword struct that simply disappears at the end of the function call. If you saw something disappear before your eyes, you'd be dumbfounded, too.

One of the solutions (as suggested in the error message), is to add the drop ability to the definition of the Sword struct, which would allow instances of this struct to disappear (be dropped). The ability to drop a valuable asset is not a desirable asset property in this case, so another solution is needed. Another way to solve this problem is to transfer ownership of the sword .

To get the test to work, we will need to use the transfer module, which is imported by default. Add the following lines to the end of the test function (after the assert! call) to transfer ownership of the sword to a freshly created dummy address:

Run the test command again. Now the output shows a single successful test has run:

Use a filter string to run only a matching subset of the unit tests. With a filter string provided, the sui move test checks the fully qualified (

::::) name for a match.

Example:

The previous command runs all tests whose name contains sword .

You can discover more testing options through:

The previous testing example uses Move but isn't specific to Sui beyond using some Sui packages, such as sui::tx_context and sui::transfer . While this style of testing is already useful for writing Move code for Sui, you might also want to test additional Sui-specific features. In particular, a Move call in Sui is encapsulated in a Sui transaction, and you might want to test interactions between different transactions within a single test (for example, one transaction creating an object and the other one transferring it).

Sui-specific testing is supported through the test_scenario module that provides Sui-related testing functionality otherwise unavailable in pure Move and its testing framework.

The test_scenario module provides a scenario that emulates a series of Sui transactions, each with a potentially different user executing them. A test using this module typically starts the first transaction using the test_scenario::begin function. This function takes an address of the user executing the transaction as its argument and returns an instance of the Scenario struct representing a scenario.

An instance of the Scenario struct contains a per-address object pool emulating Sui object storage, with helper functions provided to manipulate objects in the pool. After the first transaction finishes, subsequent test transactions start with the test_scenario::next_tx function. This function takes an instance of the Scenario struct representing the current scenario and an address of a user as arguments.

Update your example.move file to include a function callable from Sui that implements sword creation. With this in place, you can then add a multi-transaction test that uses the test_scenario module to test these new capabilities. Put this functions after the accessors (Part 5 in comments).

The code of the new functions uses struct creation and Sui-internal modules (tx_context) in a way similar to what you have seen in the previous sections. The important part is for the function to have correct signatures.

With the new function included, add another test function to make sure it behaves as expected.

There are some details of the new testing function to pay attention to. The first thing the code does is create some addresses that represent users participating in the testing scenario. The test then creates a scenario by starting the first transaction on behalf of the initial sword owner.

The initial owner then executes the second transaction (passed as an argument to the test_scenario::next_tx function), who then transfers the sword they now own to the final owner. In pure Move there is no notion of Sui storage; consequently, there is no easy way for the emulated Sui transaction to retrieve it from storage. This is where the test_scenario module helps - its take_from_sender function allows an address-owned object of a given type (Sword) executing the current transaction to be available for Move code manipulation. For now, assume that there is only one such object. In this case, the test transfers the object it retrieves from storage to another address.

Transaction effects, such as object creation and transfer become visible only after a given transaction completes. For example, if the second transaction in the running example created a sword and transferred it to the administrator's address, it would only become available for retrieval from the administrator's address (via test_scenario , take_from_sender , or take_from_address functions) in the third transaction.

The final owner executes the third and final transaction that retrieves the sword object from storage and checks if it has the expected properties. Remember, as described in Testing a package , in the pure Move testing scenario, after an object is available in Move code (after creation or retrieval from emulated storage), it cannot simply disappear.

In the pure Move testing function, the function transfers the sword object to the fake address to handle the disappearing problem. The test_scenario package provides a more elegant solution, however, which is closer to what happens when Move code actually executes in the context of Sui - the package simply returns the sword to the object pool using the test_scenario::return_to_sender function. For scenarios where returning to the sender is not desirable or if you would like to simply destroy the object, the test_utils module also provides the generic destroy function, that can be used on any type T regardless of its ability. It is advisable to check out other useful functions in the test_scenario and test_utils modules as well.

Run the test command again to see two successful tests for our module:

Each module in a package can include a special initializer function that runs at publication time. The goal of an initializer function is to pre-initialize module-specific data (for example, to create singleton objects). The initializer function must have the following properties for it to execute at publication:

For example, the following init functions are all valid:

While the sui move command does not support publishing explicitly, you can still test module initializers using the testing framework by dedicating the first transaction to executing the initializer function.

The init function for the module in the running example creates a Forge object.

The tests you have so far call the init function, but the initializer function itself isn't tested to ensure it properly creates a Forge object. To test this functionality, add a new_sword function to take the forge as a parameter and to update the number of created swords at the end of the function. If this were an actual module, you'd replace the sword_create function with new_sword . To keep the existing tests from failing, however, we will keep both functions.

Now, create a function to test the module initialization:

As the new test function shows, the first transaction (explicitly) calls the initializer. The next transaction checks if the Forge object has been created and properly initialized. Finally, the admin uses the Forge to create a sword and transfer it to the initial owner.

You can refer to the source code for the package (with all the tests and functions properly adjusted) in the first_package module in the sui/examples directory. You can also use the following toggle to review the complete code.

example.move

Testing a package

Sui includes support for the Move testing framework. Using the framework, you can write unit tests that analyze Move code much like test frameworks for other languages, such as the built-in Rust testing framework or the JUnit framework for Java.

An individual Move unit test is encapsulated in a public function that has no parameters, no return values, and has the #[test] annotation. The testing framework executes such functions when you call the sui move test command from the package root (my_move_package directory as per the current running example):

If you execute this command for the package created in Write a Package , you see the following output. Unsurprisingly, the test result has an OK status because there are no tests written yet to fail.

To actually test your code, you need to add test functions. Start with adding a basic test function to the example.move file, inside the module definition:

As the code shows, the unit test function (test_sword_create()) creates a dummy instance of the TxContext struct and assigns it to ctx . The function then creates a sword object using ctx to create a unique identifier and assigns 42 to the

magic parameter and 7 to strength . Finally, the test calls the magic and strength accessor functions to verify that they return correct values.

The function passes the dummy context, ctx , to the object::new function as a mutable reference argument (&mut), but passes sword to its accessor functions as a read-only reference argument, &sword .

Now that you have a test function, run the test command again:

After running the test command, however, you get a compilation error instead of a test result:

The error message contains all the necessary information to debug the code. The faulty code is meant to highlight one of the Move language's safety features.

The Sword struct represents a game asset that digitally mimics a real-world item. Obviously, a real sword cannot simply disappear (though it can be explicitly destroyed), but there is no such restriction on a digital one. In fact, this is exactly what's happening in the test function - you create an instance of a Sword struct that simply disappears at the end of the function call. If you saw something disappear before your eyes, you'd be dumbfounded, too.

One of the solutions (as suggested in the error message), is to add the drop ability to the definition of the Sword struct, which would allow instances of this struct to disappear (be dropped). The ability to drop a valuable asset is not a desirable asset property in this case, so another solution is needed. Another way to solve this problem is to transfer ownership of the sword .

To get the test to work, we will need to use the transfer module, which is imported by default. Add the following lines to the end of the test function (after the assert! call) to transfer ownership of the sword to a freshly created dummy address:

Run the test command again. Now the output shows a single successful test has run:

Use a filter string to run only a matching subset of the unit tests. With a filter string provided, the sui move test checks the fully qualified (

::::) name for a match.

Example:

The previous command runs all tests whose name contains sword .

You can discover more testing options through:

The previous testing example uses Move but isn't specific to Sui beyond using some Sui packages, such as sui::tx_context and sui::transfer . While this style of testing is already useful for writing Move code for Sui, you might also want to test additional Sui-specific features. In particular, a Move call in Sui is encapsulated in a Sui transaction, and you might want to test interactions between different transactions within a single test (for example, one transaction creating an object and the other one transferring it).

Sui-specific testing is supported through the test_scenario module that provides Sui-related testing functionality otherwise unavailable in pure Move and its testing framework.

The test_scenario module provides a scenario that emulates a series of Sui transactions, each with a potentially different user executing them. A test using this module typically starts the first transaction using the test_scenario::begin function. This function takes an address of the user executing the transaction as its argument and returns an instance of the Scenario struct representing a scenario.

An instance of the Scenario struct contains a per-address object pool emulating Sui object storage, with helper functions provided to manipulate objects in the pool. After the first transaction finishes, subsequent test transactions start with the test_scenario::next_tx function. This function takes an instance of the Scenario struct representing the current scenario and an address of a user as arguments.

Update your example.move file to include a function callable from Sui that implements sword creation. With this in place, you can then add a multi-transaction test that uses the test_scenario module to test these new capabilities. Put this functions after the accessors (Part 5 in comments).

The code of the new functions uses struct creation and Sui-internal modules (tx_context) in a way similar to what you have seen in the previous sections. The important part is for the function to have correct signatures.

With the new function included, add another test function to make sure it behaves as expected.

There are some details of the new testing function to pay attention to. The first thing the code does is create some addresses that represent users participating in the testing scenario. The test then creates a scenario by starting the first transaction on behalf of the initial sword owner.

The initial owner then executes the second transaction (passed as an argument to the test_scenario::next_tx function), who then transfers the sword they now own to the final owner. In pure Move there is no notion of Sui storage; consequently, there is no easy way for the emulated Sui transaction to retrieve it from storage. This is where the test_scenario module helps - its take_from_sender function allows an address-owned object of a given type (Sword) executing the current transaction to be available for Move code manipulation. For now, assume that there is only one such object. In this case, the test transfers the object it retrieves from storage to another address.

Transaction effects, such as object creation and transfer become visible only after a given transaction completes. For example, if the second transaction in the running example created a sword and transferred it to the administrator's address, it would only become available for retrieval from the administrator's address (via test_scenario , take_from_sender , or take_from_address functions) in the third transaction.

The final owner executes the third and final transaction that retrieves the sword object from storage and checks if it has the expected properties. Remember, as described in Testing a package , in the pure Move testing scenario, after an object is available in Move code (after creation or retrieval from emulated storage), it cannot simply disappear.

In the pure Move testing function, the function transfers the sword object to the fake address to handle the disappearing problem. The test_scenario package provides a more elegant solution, however, which is closer to what happens when Move code actually executes in the context of Sui - the package simply returns the sword to the object pool using the test_scenario::return_to_sender function. For scenarios where returning to the sender is not desirable or if you would like to simply destroy the object, the test_utils module also provides the generic destroy function, that can be used on any type T regardless of its ability. It is advisable to check out other useful functions in the test_scenario and test_utils modules as well.

Run the test command again to see two successful tests for our module:

Each module in a package can include a special initializer function that runs at publication time. The goal of an initializer function is to pre-initialize module-specific data (for example, to create singleton objects). The initializer function must have the following properties for it to execute at publication:

For example, the following init functions are all valid:

While the sui move command does not support publishing explicitly, you can still test module initializers using the testing framework by dedicating the first transaction to executing the initializer function.

The init function for the module in the running example creates a Forge object.

The tests you have so far call the init function, but the initializer function itself isn't tested to ensure it properly creates a Forge object. To test this functionality, add a new_sword function to take the forge as a parameter and to update the number of created swords at the end of the function. If this were an actual module, you'd replace the sword_create function with new_sword . To keep the existing tests from failing, however, we will keep both functions.

Now, create a function to test the module initialization:

As the new test function shows, the first transaction (explicitly) calls the initializer. The next transaction checks if the Forge object has been created and properly initialized. Finally, the admin uses the Forge to create a sword and transfer it to the initial owner.

You can refer to the source code for the package (with all the tests and functions properly adjusted) in the first_package module in the sui/examples directory. You can also use the following toggle to review the complete code.

example.move

Sui-specific testing

The previous testing example uses Move but isn't specific to Sui beyond using some Sui packages, such as sui::tx_context and sui::transfer . While this style of testing is already useful for writing Move code for Sui, you might also want to test

additional Sui-specific features. In particular, a Move call in Sui is encapsulated in a Sui transaction, and you might want to test interactions between different transactions within a single test (for example, one transaction creating an object and the other one transferring it).

Sui-specific testing is supported through the test_scenario module that provides Sui-related testing functionality otherwise unavailable in pure Move and its testing framework.

The test_scenario module provides a scenario that emulates a series of Sui transactions, each with a potentially different user executing them. A test using this module typically starts the first transaction using the test_scenario::begin function. This function takes an address of the user executing the transaction as its argument and returns an instance of the Scenario struct representing a scenario.

An instance of the Scenario struct contains a per-address object pool emulating Sui object storage, with helper functions provided to manipulate objects in the pool. After the first transaction finishes, subsequent test transactions start with the test_scenario::next_tx function. This function takes an instance of the Scenario struct representing the current scenario and an address of a user as arguments.

Update your example.move file to include a function callable from Sui that implements sword creation. With this in place, you can then add a multi-transaction test that uses the test_scenario module to test these new capabilities. Put this functions after the accessors (Part 5 in comments).

The code of the new functions uses struct creation and Sui-internal modules (tx_context) in a way similar to what you have seen in the previous sections. The important part is for the function to have correct signatures.

With the new function included, add another test function to make sure it behaves as expected.

There are some details of the new testing function to pay attention to. The first thing the code does is create some addresses that represent users participating in the testing scenario. The test then creates a scenario by starting the first transaction on behalf of the initial sword owner.

The initial owner then executes the second transaction (passed as an argument to the test_scenario::next_tx function), who then transfers the sword they now own to the final owner. In pure Move there is no notion of Sui storage; consequently, there is no easy way for the emulated Sui transaction to retrieve it from storage. This is where the test_scenario module helps - its take_from_sender function allows an address-owned object of a given type (Sword) executing the current transaction to be available for Move code manipulation. For now, assume that there is only one such object. In this case, the test transfers the object it retrieves from storage to another address.

Transaction effects, such as object creation and transfer become visible only after a given transaction completes. For example, if the second transaction in the running example created a sword and transferred it to the administrator's address, it would only become available for retrieval from the administrator's address (via test_scenario , take_from_sender , or take_from_address functions) in the third transaction.

The final owner executes the third and final transaction that retrieves the sword object from storage and checks if it has the expected properties. Remember, as described in Testing a package , in the pure Move testing scenario, after an object is available in Move code (after creation or retrieval from emulated storage), it cannot simply disappear.

In the pure Move testing function, the function transfers the sword object to the fake address to handle the disappearing problem. The test_scenario package provides a more elegant solution, however, which is closer to what happens when Move code actually executes in the context of Sui - the package simply returns the sword to the object pool using the test_scenario::return_to_sender function. For scenarios where returning to the sender is not desirable or if you would like to simply destroy the object, the test_utils module also provides the generic destroy function, that can be used on any type T regardless of its ability. It is advisable to check out other useful functions in the test_scenario and test_utils modules as well.

Run the test command again to see two successful tests for our module:

Each module in a package can include a special initializer function that runs at publication time. The goal of an initializer function is to pre-initialize module-specific data (for example, to create singleton objects). The initializer function must have the following properties for it to execute at publication:

For example, the following init functions are all valid:

While the sui move command does not support publishing explicitly, you can still test module initializers using the testing framework by dedicating the first transaction to executing the initializer function.

The init function for the module in the running example creates a Forge object.

The tests you have so far call the init function, but the initializer function itself isn't tested to ensure it properly creates a Forge object. To test this functionality, add a new_sword function to take the forge as a parameter and to update the number of created swords at the end of the function. If this were an actual module, you'd replace the sword_create function with new_sword . To keep the existing tests from failing, however, we will keep both functions.

Now, create a function to test the module initialization:

As the new test function shows, the first transaction (explicitly) calls the initializer. The next transaction checks if the Forge object has been created and properly initialized. Finally, the admin uses the Forge to create a sword and transfer it to the initial owner.

You can refer to the source code for the package (with all the tests and functions properly adjusted) in the first_package module in the sui/examples directory. You can also use the following toggle to review the complete code.

example.move

Related links