Sui Object Display

The Sui Object Display standard is a template engine that enables on-chain management of off-chain representation (display) for a type. With it, you can substitute data for an object into a template string. The standard doesn't limit the fields you can set. You can use the {property} syntax to access all object properties, and then insert them as a part of the template string.

Use a Publisher object that you own to set sui::display for a type. For more information about Publisher objects, see Publisher topic in Sui Move by Example .

In Sui Move, Display represents an object that specifies a set of named templates for the type T . For example, for a type 0x2::capy::Capy the display syntax is: Display<0x2::capy::Capy> .

Sui Full nodes process all objects of the type T by matching the Display definition, and return the processed result when you query an object with the { showDisplay: true } setting in the query.

The basic set of properties suggested includes:

The following code sample demonstrates how the Display for an example Hero module varies based on the name , id , and image_url properties of the type Hero . The following represents the template the init function defines:

The display::new call creates a Display , either in a custom function or module initializer, or as part of a programmable transaction. The following code sample demonstrates how to create a Display :

After you create the Display , you can modify it. The following code sample demonstrates how to modify a Display :

Next, the update_version call applies the changes and sets the Display for the T by emitting an event. Full nodes receive the event and use the data in the event to retrieve a template for the type.

The following code sample demonstrates how to use the update_version call:

In Sui, utility objects enable authorization for capabilities. Almost all modules have features that can be accessed only with the required capability. Generic modules allow one capability per application, such as a marketplace. Some capabilities mark ownership of a shared object on-chain, or access the shared data from another account. With capabilities, it is important to provide a meaningful description of objects to facilitate user interface implementation. This helps avoid accidentally transferring the wrong object when objects are similar. It also provides a user-friendly description of items that users see.

The following example demonstrates how to create a capy capability:

A common case with in-game items is to have a large number of similar objects grouped by some criteria. It is important to optimize their size and the cost to mint and update them. Typically, a game uses a single source image or URL per group or item criteria. Storing the source image inside of every object is not optimal. In some cases, users mint in-game items when a game allows them or when they purchase an in-game item. To enable this, some IPFS/Arweave metadata must be created and stored in advance. This requires additional logic that is usually not related to the in-game properties of the item.

The following example demonstrates how to create a Capy:

Sui Capys use dynamic image generation. When a Capy is born, its attributes determine the Capy's appearance, such as color or pattern. When a user puts an item on a Capy, the Capy's appearance changes. When users put multiple items on a Capy, there's a chance of a bonus for a combination of items.

To implement this, the Capys game API service refreshes the image in response to a user-initiated change. The URL for a Capy is a template with the capy.id . But storing the full URL - as well as other fields in the Capy object due to their diverse population - also leads to users paying for excess storage and increased gas fees.

The following example demonstrates how to implement dynamic image generation:

This is the simplest scenario - an object represents everything itself. It is very easy to apply a metadata standard to an object of this kind, especially if the object stays immutable forever. However, if the metadata standard evolves and some ecosystem projects add new features for some properties, this object always stays in its original form and might require backward-compatible changes.

Display properties

The basic set of properties suggested includes:

The following code sample demonstrates how the Display for an example Hero module varies based on the name , id , and image_url properties of the type Hero . The following represents the template the init function defines:

The display::new call creates a Display , either in a custom function or module initializer, or as part of a programmable transaction. The following code sample demonstrates how to create a Display :

After you create the Display , you can modify it. The following code sample demonstrates how to modify a Display :

Next, the update_version call applies the changes and sets the Display for the T by emitting an event. Full nodes receive the event and use the data in the event to retrieve a template for the type.

The following code sample demonstrates how to use the update_version call:

In Sui, utility objects enable authorization for capabilities. Almost all modules have features that can be accessed only with the required capability. Generic modules allow one capability per application, such as a marketplace. Some capabilities mark ownership of a shared object on-chain, or access the shared data from another account. With capabilities, it is important to provide a meaningful description of objects to facilitate user interface implementation. This helps avoid accidentally transferring the wrong object when objects are similar. It also provides a user-friendly description of items that users see.

The following example demonstrates how to create a capy capability:

A common case with in-game items is to have a large number of similar objects grouped by some criteria. It is important to optimize their size and the cost to mint and update them. Typically, a game uses a single source image or URL per group or item criteria. Storing the source image inside of every object is not optimal. In some cases, users mint in-game items when a game allows them or when they purchase an in-game item. To enable this, some IPFS/Arweave metadata must be created and stored in advance. This requires additional logic that is usually not related to the in-game properties of the item.

The following example demonstrates how to create a Capy:

Sui Capys use dynamic image generation. When a Capy is born, its attributes determine the Capy's appearance, such as color or pattern. When a user puts an item on a Capy, the Capy's appearance changes. When users put multiple items on a Capy, there's a chance of a bonus for a combination of items.

To implement this, the Capys game API service refreshes the image in response to a user-initiated change. The URL for a Capy is a template with the capy.id . But storing the full URL - as well as other fields in the Capy object due to their diverse population - also leads to users paying for excess storage and increased gas fees.

The following example demonstrates how to implement dynamic image generation:

This is the simplest scenario - an object represents everything itself. It is very easy to apply a metadata standard to an object of this kind, especially if the object stays immutable forever. However, if the metadata standard evolves and some ecosystem projects add new features for some properties, this object always stays in its original form and might require backward-compatible changes.

Work with Object Display

The display::new call creates a Display , either in a custom function or module initializer, or as part of a programmable transaction. The following code sample demonstrates how to create a Display :

After you create the Display , you can modify it. The following code sample demonstrates how to modify a Display :

Next, the update_version call applies the changes and sets the Display for the T by emitting an event. Full nodes receive the event and use the data in the event to retrieve a template for the type.

The following code sample demonstrates how to use the update_version call:

In Sui, utility objects enable authorization for capabilities. Almost all modules have features that can be accessed only with the required capability. Generic modules allow one capability per application, such as a marketplace. Some capabilities mark ownership of a shared object on-chain, or access the shared data from another account. With capabilities, it is important to provide a meaningful description of objects to facilitate user interface implementation. This helps avoid accidentally transferring the wrong object when objects are similar. It also provides a user-friendly description of items that users see.

The following example demonstrates how to create a capy capability:

A common case with in-game items is to have a large number of similar objects grouped by some criteria. It is important to optimize

their size and the cost to mint and update them. Typically, a game uses a single source image or URL per group or item criteria. Storing the source image inside of every object is not optimal. In some cases, users mint in-game items when a game allows them or when they purchase an in-game item. To enable this, some IPFS/Arweave metadata must be created and stored in advance. This requires additional logic that is usually not related to the in-game properties of the item.

The following example demonstrates how to create a Capy:

Sui Capys use dynamic image generation. When a Capy is born, its attributes determine the Capy's appearance, such as color or pattern. When a user puts an item on a Capy, the Capy's appearance changes. When users put multiple items on a Capy, there's a chance of a bonus for a combination of items.

To implement this, the Capys game API service refreshes the image in response to a user-initiated change. The URL for a Capy is a template with the capy.id . But storing the full URL - as well as other fields in the Capy object due to their diverse population - also leads to users paying for excess storage and increased gas fees.

The following example demonstrates how to implement dynamic image generation:

This is the simplest scenario - an object represents everything itself. It is very easy to apply a metadata standard to an object of this kind, especially if the object stays immutable forever. However, if the metadata standard evolves and some ecosystem projects add new features for some properties, this object always stays in its original form and might require backward-compatible changes.

Sui utility objects

In Sui, utility objects enable authorization for capabilities. Almost all modules have features that can be accessed only with the required capability. Generic modules allow one capability per application, such as a marketplace. Some capabilities mark ownership of a shared object on-chain, or access the shared data from another account. With capabilities, it is important to provide a meaningful description of objects to facilitate user interface implementation. This helps avoid accidentally transferring the wrong object when objects are similar. It also provides a user-friendly description of items that users see.

The following example demonstrates how to create a capy capability:

A common case with in-game items is to have a large number of similar objects grouped by some criteria. It is important to optimize their size and the cost to mint and update them. Typically, a game uses a single source image or URL per group or item criteria. Storing the source image inside of every object is not optimal. In some cases, users mint in-game items when a game allows them or when they purchase an in-game item. To enable this, some IPFS/Arweave metadata must be created and stored in advance. This requires additional logic that is usually not related to the in-game properties of the item.

The following example demonstrates how to create a Capy:

Sui Capys use dynamic image generation. When a Capy is born, its attributes determine the Capy's appearance, such as color or pattern. When a user puts an item on a Capy, the Capy's appearance changes. When users put multiple items on a Capy, there's a chance of a bonus for a combination of items.

To implement this, the Capys game API service refreshes the image in response to a user-initiated change. The URL for a Capy is a template with the capy.id . But storing the full URL - as well as other fields in the Capy object due to their diverse population - also leads to users paying for excess storage and increased gas fees.

The following example demonstrates how to implement dynamic image generation:

This is the simplest scenario - an object represents everything itself. It is very easy to apply a metadata standard to an object of this kind, especially if the object stays immutable forever. However, if the metadata standard evolves and some ecosystem projects add new features for some properties, this object always stays in its original form and might require backward-compatible changes.

Typical objects with data duplication

A common case with in-game items is to have a large number of similar objects grouped by some criteria. It is important to optimize their size and the cost to mint and update them. Typically, a game uses a single source image or URL per group or item criteria. Storing the source image inside of every object is not optimal. In some cases, users mint in-game items when a game allows them or when they purchase an in-game item. To enable this, some IPFS/Arweave metadata must be created and stored in advance. This requires additional logic that is usually not related to the in-game properties of the item.

The following example demonstrates how to create a Capy:

Sui Capys use dynamic image generation. When a Capy is born, its attributes determine the Capy's appearance, such as color or pattern. When a user puts an item on a Capy, the Capy's appearance changes. When users put multiple items on a Capy, there's a chance of a bonus for a combination of items.

To implement this, the Capys game API service refreshes the image in response to a user-initiated change. The URL for a Capy is a template with the capy.id . But storing the full URL - as well as other fields in the Capy object due to their diverse population - also leads to users paying for excess storage and increased gas fees.

The following example demonstrates how to implement dynamic image generation:

This is the simplest scenario - an object represents everything itself. It is very easy to apply a metadata standard to an object of this kind, especially if the object stays immutable forever. However, if the metadata standard evolves and some ecosystem projects add new features for some properties, this object always stays in its original form and might require backward-compatible changes.

Unique objects with dynamic representation

Sui Capys use dynamic image generation. When a Capy is born, its attributes determine the Capy's appearance, such as color or pattern. When a user puts an item on a Capy, the Capy's appearance changes. When users put multiple items on a Capy, there's a chance of a bonus for a combination of items.

To implement this, the Capys game API service refreshes the image in response to a user-initiated change. The URL for a Capy is a template with the capy.id . But storing the full URL - as well as other fields in the Capy object due to their diverse population - also leads to users paying for excess storage and increased gas fees.

The following example demonstrates how to implement dynamic image generation:

This is the simplest scenario - an object represents everything itself. It is very easy to apply a metadata standard to an object of this kind, especially if the object stays immutable forever. However, if the metadata standard evolves and some ecosystem projects add new features for some properties, this object always stays in its original form and might require backward-compatible changes.

Objects with unique static content

This is the simplest scenario - an object represents everything itself. It is very easy to apply a metadata standard to an object of this kind, especially if the object stays immutable forever. However, if the metadata standard evolves and some ecosystem projects add new features for some properties, this object always stays in its original form and might require backward-compatible changes.