UI delegation

OpenAssetIO allows applications to provide a bespoke yet natively integrated user interface for interacting with asset management systems in a pipeline. This is accomplished though *UI delegation*. A host application that wishes to present a user with a graphical interface to interact with an asset management system can request the appropriate UI element from a plugin, and be notified by the plugin when the user has performed a relevant interaction. For example, an asset browser in place of a standard file browser.

Preamble

This section establishes some boilerplate that will be useful for the rest of the notebook.

First some standard OpenAssetIO boilerplate - see the Hello OpenAssetIO notebook for more details.

In [1]:
```
try:
    import openassetio
    import openassetio_mediacreation
except ImportError:
    print(
        "This notebook requires the packages listed in `resources/requirements.txt` to be installed"
    )
    raise

from openassetio.hostApi import HostInterface
from openassetio.log import ConsoleLogger, SeverityFilter


class NotebookHostInterface(HostInterface):
    def identifier(self):
        return "org.jupyter.notebook"

    def displayName(self):
        return "Jupyter Notebook"


host_interface = NotebookHostInterface()
logger = SeverityFilter(ConsoleLogger())
```

For demonstration purposes, we'll create a simple UI framework, making use of Jupyter's `ipywidgets` module.

In [2]:
```
import ipywidgets
from IPython.display import display


class WidgetBase:
    def __init__(self, on_value_change_cb):
        self.__on_value_change_cb = on_value_change_cb
        self._ipywidget = None

    def attach(self):
        display(self._ipywidget)
        self._ipywidget.observe(self.__observer, names="value")

    def __observer(self, change):
        self.__on_value_change_cb(change["new"])


class TextBoxWidget(WidgetBase):
    def __init__(self, on_value_change_cb, placeholder="Enter text..."):
        super().__init__(on_value_change_cb)
        self._ipywidget = ipywidgets.Text(placeholder=placeholder, continuous_update=False)

    def set_text(self, text):
        self._ipywidget.value = text

    def get_text(self):
        return self._ipywidget.value


class LabelWidget(WidgetBase):
    def __init__(self, label):
        super().__init__(lambda _unused: None)
        self._ipywidget = ipywidgets.Label(label)

    def set_text(self, text):
        self._ipywidget.value = text


class DropdownWidget(WidgetBase):
    def __init__(self, on_value_changed_cb, options):
        super().__init__(on_value_changed_cb)
```

```
        self._ipywidget = ipywidgets.Dropdown(options=options)


class ContainerWidget(list):
    def attach(self):
        for widget in self:
            widget.attach()
```

Although we're dealing mostly with the UI delegate system in this notebook, we'll need a (basic) manager plugin too. Initially only to get a `Context`, but ultimately we'll need to illustrate how UI delegate and manager plugins may communicate.

In [3]:
```
from openassetio.managerApi import ManagerInterface


class NotebookManagerInterface(ManagerInterface):
    """
    An absolutely minimal manager implementation.
    """

    def entityTraits(
            self,
            entityReferences,
            entityTraitsAccess,
            context,
            hostSession,
            successCallback,
            errorCallback,
    ):
        for idx, entity_ref in enumerate(entityReferences):
            successCallback(idx, set())

    def identifier(self):
        return "org.openassetio.example.notebook"

    def displayName(self):
        return "Notebook Manager"

    def hasCapability(self, capability):
        return capability in (
            ManagerInterface.Capability.kEntityReferenceIdentification,
            ManagerInterface.Capability.kManagementPolicyQueries,
            ManagerInterface.Capability.kEntityTraitIntrospection,
        )

    def isEntityReferenceString(self, someString, hostSession):
        return someString.startswith("notebook://")

    def managementPolicy(self, traitSets, access, context, hostSession):
        return [TraitsData() for _ in traitSets]
```

In order to use this plugin, we need a plugin system to load it. Usually, a host would use the built-in `Hybrid` / `Python` / `CppPluginSystemManagerImplementationFactory`. However, for the purposes of this notebook we'll define a custom plugin system that will simply return an instance of our `NotebookManagerInterface` defined above.

In [4]:
```
from openassetio.hostApi import ManagerImplementationFactoryInterface


class NotebookPluginSystemManagerImplementationFactory(ManagerImplementationFactoryInterface):

    def identifiers(self):
        return ["org.openassetio.example.notebook"]

    def instantiate(self, identifier):
        assert identifier == "org.openassetio.example.notebook"
        return NotebookManagerInterface()
```

With the notebook plugin system now available, we can use it to construct a `Manager` instance and from that retrieve a `Context`. We'll wrap the manager creation in a function for reuse later.

In [5]:
```
from openassetio.hostApi import ManagerFactory


def make_manager():
    manager = ManagerFactory.createManagerForInterface(
        "org.openassetio.example.notebook",
        host_interface,
        NotebookPluginSystemManagerImplementationFactory(logger),
        logger,
    )
    manager.initialize({})
    return manager
```

```
manager = make_manager()
context = manager.createContext()
```

Hello UI

Let's start by defining the most minimal UI delegate plugin possible. The following UI delegate doesn't do anything interesting - it simply refuses any request.

In [6]:
```
from typing import Optional

from openassetio import Context
from openassetio.managerApi import HostSession
from openassetio.trait import TraitsData
from openassetio.ui.access import UIAccess
from openassetio.ui.managerApi import UIDelegateInterface, UIDelegateRequest, \
    UIDelegateStateInterface


InfoDictionary = dict[str, int | float | str | bool]


class NotebookUIDelegateInterface(UIDelegateInterface):
    def identifier(self):
        return "org.openassetio.example.notebook"

    def displayName(self):
        return "Notebook UI delegate"

    def info(self) -> InfoDictionary:
        return super().info()  # <- adds `"isPython": True`

    def initialize(self, uiDelegateSettings: InfoDictionary, hostSession: HostSession):
        pass

    def close(self, hostSession: HostSession):
        pass

    def settings(self, hostSession: HostSession) -> InfoDictionary:
        return {}

    def uiPolicy(
            self, uiTraitSet: set[str], uiAccess: UIAccess, context: Context,
            hostSession: HostSession
    ) -> TraitsData:
        policy = TraitsData()
        return policy

    def populateUI(
            self,
            uiTraitsData: TraitsData,
            uiAccess: UIAccess,
            uiRequest: UIDelegateRequest,
            context: Context,
            hostSession: HostSession,
    ) -> Optional[UIDelegateStateInterface]:
        return None
```

We have several methods that must be overridden from the base `UIDelegateInterface` class. Most of them mirror a method in manager plugins (`ManagerInterface`), and have similar usage.

The `identifier` *must* match the identifier of the corresponding manager plugin. It is this that is used to link the two plugins together, in particular when loading the plugin.

`displayName` may be used by host applications for logging, plugin selection, etc.

`info` may be used to communicate arbitrary metadata about the plugin as a simple string key - primitive value dictionary. Certain keys have a special meaning. For example `kInfoKey_IsPython` (`"isPython"`) indicates that the plugin is written in pure Python (this particular key is added to the `info()` in the default base class implementation for Python plugins).

`initialize` must be called before the plugin can be used. It is called automatically by `UIDelegateFactory.defaultManagerForInterface()` - the `uiDelegateSettings` in this case will be the same settings as provided to the corresponding manager plugin, from the file specified by the `OPENASSETIO_DEFAULT_CONFIG` environment variable.

`close` instructs the UI delegate to dispose of all held references and clean up ready for destruction. It is called automatically when the corresponding `UIDelegate` middleware instance, held by the host application, is destroyed. However, it can be called explicitly by the host to clean up and re-use the UI delegate.

`settings` returns the current settings applied to the UI delegate (including any default values). These settings should reflect those passed in to `initialize` .

`uiPolicy` is the UI delegation equivalent of a manager plugin's `managementPolicy` method. It allows hosts to introspect the capabilities and preferences of the plugin ahead of time. Note that a positive response from `uiPolicy` is necessary, but not sufficient, to guarantee a UI delegation request will be supported.

`populateUI` is where the magic happens. It is this method that will be called to instantiate UI elements and set up a communication channel with the host. A return value of `None` indicates that UI delegation is not supported for the given arguments.

Since the above UI delegate plugin won't do anything interesting, let's redefine the `populateUI` method to render a text box.

In [7]:
```
class NotebookUIDelegateInterface(NotebookUIDelegateInterface):
    def populateUI(self, uiTraitsData, uiAccess, request, context, hostSession):
        def on_value_changed_cb(_some_text):
            pass

        TextBoxWidget(on_value_changed_cb).attach()

        return UIDelegateStateInterface()
```

Note that we now return a `UIDelegateStateInterface()` object. This will signal to the host that the request was accepted. The `UIDelegateStateInterface` class (or rather, subclasses of it) allows information to be communicated back to the host application, and facilitates setting up a communication channel, as we will see shortly.

In order to use this plugin, we need a plugin system to load it. Usually, a host would use the built-in `Hybrid` / `Python` / `CppPluginSystemUIDelegateImplementationFactory`, in exactly the same way as for manager plugins (via a `Hybrid` / `Python` / `CppPluginSystemManagerImplementationFactory`). However, for the purposes of this notebook we'll define a custom plugin system that will simply return an instance of our `NotebookUIDelegateInterface` defined above.

In [8]:
```
from openassetio.ui.hostApi import UIDelegateImplementationFactoryInterface


class NotebookPluginSystemUIDelegateFactory(UIDelegateImplementationFactoryInterface):

    def identifiers(self):
        return ["org.openassetio.example.notebook"]

    def instantiate(self, identifier):
        assert identifier == "org.openassetio.example.notebook"
        return NotebookUIDelegateInterface()
```

Now we have this, let's put on our host application hat, and instantiate the plugin. We'll wrap the UI delegate creation in a function for reuse later.

In [9]:
```
from openassetio.ui.hostApi import UIDelegateFactory


def make_ui_delegate():
    ui_delegate = UIDelegateFactory.createUIDelegateForInterface(
        "org.openassetio.example.notebook",
        host_interface,
        NotebookPluginSystemUIDelegateFactory(logger),
        logger,
    )
    # We must be sure to `initialize` with relevant settings. This is
    # done automatically if `defaultUIDelegateForInterface()` was used
    # instead, using the settings from the `OPENASSETIO_DEFAULT_CONFIG`
    # file (same as for manager plugins).
    ui_delegate.initialize({})
    return ui_delegate


ui_delegate = make_ui_delegate()
```

Now let's delegate some UI!

In [10]:
```
from openassetio.trait import TraitsData
from openassetio.ui.access import UIAccess
from openassetio.ui.hostApi import UIDelegateRequestInterface


ui_traits_data = TraitsData()
ui_request = UIDelegateRequestInterface()

initial_state = ui_delegate.populateUI(ui_traits_data, UIAccess.kRead, ui_request, context)
```

Enter text...

Success! Note that the call to the UI delegate is parameterised on UI-specific traits, an access mode, and a `UIDelegateRequestInterface` object. The `UIDelegateRequestInterface` object is the host-side version of the `UIDelegateStateInterface` object. It allows information to be communicated from the host application to the UI delegate. Together

the `UIDelegateRequestInterface` and `UIDelegateStateInterface` (or rather, subclasses of these) classes provide the primary means of back-and-forth communication, as we will see shortly.

Note that the "other side" of the API receives a middleware type wrapping the interface type. That is, the UI delegate plugin receives a `UIDelegateRequest` object that wraps the `UIDelegateRequestInterface` provided by the host; and the host receives a `UIDelegateState` object that wraps the `UIDelegateStateInterface` provided by the UI delegate plugin. The middleware types have the same accessors as the interface types, but allow additional validation and other functionality to be injected by OpenAssetIO.

In particular, the `UIDelegateRequest` exposes `entityReferences()`, `entityTraitsDatas()`, `nativeData()` and `stateChangedCallback()` methods, whereas the `UIDelegateState` exposes `entityReferences()`, `entityTraitsDatas()`, `nativeData()` and `updateRequestCallback()` methods. These will be discussed in turn in the following sections.

Clearly the above widget still isn't very useful. For starters, we currently have no way to extract the value that the user inputs into the widget.

stateChangedCallback

Since the widget is an opaque black box as far as the host is concerned, we need to ask the plugin to communicate salient information back to the host whenever the plugin feels it is relevant to do so. For this we'll augment our `UIDelegateRequestInterface` instance with a callback.

So let's assume a callback is provided and redefine the `populateUI` method to use it. The callback takes a `UIDelegateStateInterface`, which is a base class that needs implementing by the UI delegate. We'll build up a simple implementation, starting with a way to provide a list of entity references.

In [11]:
```
from openassetio import EntityReference


class NotebookUIDelegateState(UIDelegateStateInterface):
    """
    Initial implementation just supports getting/setting a list of
    entity references.
    """

    def __init__(self):
        UIDelegateStateInterface.__init__(self)
        self.__entity_references = []

    def entityReferences(self):
        # Override base class, which just returns an empty list.
        return self.__entity_references

    def setEntityReferences(self, entity_references):
        self.__entity_references = entity_references


class NotebookUIDelegateInterface(NotebookUIDelegateInterface):
    def populateUI(self, uiTraitsData, uiAccess, request, context, hostSession):
        current_state = NotebookUIDelegateState()

        def on_value_changed_cb(some_text):
            if state_changed_cb := request.stateChangedCallback():
                current_state.setEntityReferences([EntityReference("notebook://" + some_text)])
                state_changed_cb(current_state)

        TextBoxWidget(on_value_changed_cb).attach()

        return current_state
```

We see that the `UIDelegateRequest` object can carry a `stateChangedCallback`, which is a function of a single parameter, a `UIDelegateStateInterface` instance. The UI delegate plugin should fill in as much information as it can in the `UIDelegateStateInterface`. We'll examine the other properties of this class later.

Let's now make use of this redefined plugin on the host side. Similar to the `UIDelegateStateInterface`, the `UIDelegateRequestInterface` class is just a stub by default and needs subclassing to provide its functionality.

In [12]:
```
class NotebookHostUIRequest(UIDelegateRequestInterface):
    """
    Initial implementation just supports setting a callback for state
    changes.
    """

    def __init__(self, stateChangedCallback=None):
        UIDelegateRequestInterface.__init__(self)
        self.__state_changed_cb = stateChangedCallback

    def stateChangedCallback(self):
        # Override base class implementation, which just returns None.
        return self.__state_changed_cb
```

```
def closure():
    ui_delegate = make_ui_delegate()  # Must make a new instance since we redefined the class.

    instructions = ipywidgets.Output()
    entity_refs_output = ipywidgets.Output()

    with instructions:
        print("Type some text and press ENTER")
    display(instructions)

    def state_changed_cb(new_state):
        entity_refs_output.clear_output()
        with entity_refs_output:
            print(f"From UI delegate: {new_state.entityReferences()}")

    ui_request = NotebookHostUIRequest(stateChangedCallback=state_changed_cb)

    _initial_state = ui_delegate.populateUI(ui_traits_data, UIAccess.kRead, ui_request, context)

    display(entity_refs_output)


closure()
```

```
Type some text and press ENTER
```

some_entity

```
From UI delegate: [EntityReference('notebook://some_entity')]
```

If viewing this notebook in Jupyter, we can input some text in the box and hit ENTER (or defocus the text box), and see a list of entity references returned from the UI delegate plugin, constructed from the given user input!

nativeData

Typically, a host application will consist of a complex UI hierarchy, and we want to place the UI delegate widgets in a specific location in that hierarchy. So far, our UI delegate is just blindly rendering the widget immediately on calling `populateUI`.

For example, the host application may want to retrieve a "detached" widget, which the host can then place in the hierarchy itself. Alternatively, the host may have a UI panel that it wishes the plugin to populate. Or perhaps the application uses an immediate-mode UI framework, where the "widget" is actually a function that is called every frame...

In order to cover the maximum number of use cases, two `nativeData` objects are provided - one in the `UIDelegateRequest` and one in the `UIDelegateState`. These can be any arbitrary object, and it is up to the host application to document its expectations for how these are used.

Let's make a slightly more complex host UI and use `nativeData` to get a widget detached from the hierarchy until the host wishes to render it.

We'll first need to add support for `nativeData` to our state, and redefine `populateUI` to use it

In [13]:
```
class NotebookUIDelegateState(NotebookUIDelegateState):
    """
    Redefine to add support for nativeData.
    """

    def __init__(self):
        super().__init__()
        self.__native_data = None

    def nativeData(self):
        # Override base class, which just returns None.
        return self.__native_data

    def setNativeData(self, native_data):
        self.__native_data = native_data


class NotebookUIDelegateInterface(NotebookUIDelegateInterface):
    def populateUI(self, uiTraitsData, uiAccess, request, context, hostSession):
        current_state = NotebookUIDelegateState()

        def on_value_changed_cb(some_text):
            if state_changed_cb := request.stateChangedCallback():
                current_state.setEntityReferences([EntityReference("notebook://" + some_text)])
                state_changed_cb(current_state)

        widget = TextBoxWidget(on_value_changed_cb)  # No longer calling `.attach()`

        current_state.setNativeData(widget)
```

```
        return current_state
```

Now we can construct a host UI, with the delegated widget inserted in the correct location

In [14]:
```
def closure():
    ui_delegate = make_ui_delegate()

    entity_ref_output_label = LabelWidget("")

    def state_changed_cb(new_state):
        entity_ref_output_label.set_text(f"From UI delegate: {new_state.entityReferences()}")

    ui_request = NotebookHostUIRequest(stateChangedCallback=state_changed_cb)

    initial_state = ui_delegate.populateUI(ui_traits_data, UIAccess.kRead, ui_request, context)

    container = ContainerWidget()
    container.append(LabelWidget("Type some text and press ENTER"))
    container.append(initial_state.nativeData())
    container.append(entity_ref_output_label)
    container.attach()


closure()
```

Type some text and press ENTER

some_entity

From UI delegate: [EntityReference('notebook://some_entity')]

UI traits

Now we can request a new "detached" widget, to be placed in the UI hierarchy by the host. But what if we want the UI delegate to populate a pre-existing UI element instead? Or, perhaps we want both - we want the UI delegate to populate a container *and* return the newly created element?

To solve this we'll make use of the `nativeData` on the `UIDelegateRequest`, and instruct the UI delegate's behaviour by populating the `uiTraitsData` argument.

The `uiTraitsData` holds OpenAssetIO *traits*. Traits are used for various purposes across OpenAssetIO - for defining the qualities and properties of entities and their relationships, for defining management policies, and here, for defining the desired qualities and behaviour of the delegated UI element.

In [15]:
```
from openassetio_mediacreation.traits.ui import DetachedTrait, InPlaceTrait, SingularTrait


class NotebookUIDelegateInterface(NotebookUIDelegateInterface):
    def populateUI(self, uiTraitsData, uiAccess, request, context, hostSession):

        # <===== New code here
        if not SingularTrait.isImbuedTo(uiTraitsData):
            # We can only supply a single entity reference.
            return None
        # ==================>

        current_state = NotebookUIDelegateState()

        def on_value_changed_cb(some_text):
            if state_changed_cb := request.stateChangedCallback():
                current_state.setEntityReferences([EntityReference("notebook://" + some_text)])
                state_changed_cb(current_state)

        widget = TextBoxWidget(on_value_changed_cb)

        # <===== New code here
        if DetachedTrait.isImbuedTo(uiTraitsData):
            # Host wants a reference to the new widget.
            current_state.setNativeData(widget)

        if InPlaceTrait.isImbuedTo(uiTraitsData):
            # Host wants a container to be populated.
            container = request.nativeData()
            container.append(widget)
        # ==================>

        return current_state
```

The `SingularTrait` indicates that the host only wants or provides a single element, in this case an entity reference. The `InPlaceTrait` and `DetachedTrait` are used above to modify the native behaviour of the UI delegate. By adding more traits we can be even more specific about the kind of UI that we want.

So far we can only populate the UI with a single type of widget - an inline entity-providing widget (in this case a simple text box, but it could be something more imaginative). What if we wanted to instead display an asset browser (e.g. to replace a typical File->Open menu dialog)? Again, we can use traits to instruct the UI delegate.

In [16]:
```
from openassetio_mediacreation.traits.ui import EntityProviderTrait, InlineTrait, BrowserTrait


class NotebookUIDelegateInterface(NotebookUIDelegateInterface):
    def populateUI(self, uiTraitsData, uiAccess, request, context, hostSession):

        if not SingularTrait.isImbuedTo(uiTraitsData):
            # We can only supply a single entity reference.
            return None

        current_state = NotebookUIDelegateState()

        def on_value_changed_cb(some_text):
            if state_changed_cb := request.stateChangedCallback():
                current_state.setEntityReferences([EntityReference("notebook://" + some_text)])
                state_changed_cb(current_state)

        # <===== New code here
        ui_trait_set = uiTraitsData.traitSet()

        if {EntityProviderTrait.kId, InlineTrait.kId} <= ui_trait_set:
            widget = TextBoxWidget(on_value_changed_cb)
        elif {EntityProviderTrait.kId, BrowserTrait.kId} <= ui_trait_set:
            widget = DropdownWidget(on_value_changed_cb, ["first/entity", "second/entity"])
        else:
            # We don't support other kinds of UI.
            return None
        # ====================>

        if DetachedTrait.isImbuedTo(uiTraitsData):
            # Host wants a reference to the new widget.
            current_state.setNativeData(widget)

        if InPlaceTrait.isImbuedTo(uiTraitsData):
            # Host wants a container to be populated.
            container = request.nativeData()
            container.append(widget)

        return current_state
```

Above, we use a simple dropdown widget as our "asset browser", but typically this would be a much more complex UI element. The `EntityProviderTrait` indicates that host wants the UI delegate to provide entity references. The `InlineTrait` indicates that the widget should be compact enough to share its container with other widgets, typically on a single row. The `BrowserTrait` indicates that the widget should guide the user through multiple options.

Let's use this new functionality on the host side. For illustration, we'll request a detached widget for the browser, and mutate an existing container in-place for the inline box.

In [17]:
```
class NotebookHostUIRequest(NotebookHostUIRequest):
    """
    Redefine to add support for nativeData.
    """

    def __init__(self, nativeData=None, **kwargs):
        super().__init__(**kwargs)
        self.__native_data = nativeData

    def nativeData(self):
        # Override base class implementation, which just returns None.
        return self.__native_data


def closure():
    ui_delegate = make_ui_delegate()
    container = ContainerWidget()

    ######################################################################
    # First lets display the "asset browser".

    # UI traits for an entity browser.
    ui_traits_data = TraitsData({EntityProviderTrait.kId, SingularTrait.kId, BrowserTrait.kId})
    # We'll request a "detached" widget for the browser.
    DetachedTrait.imbueTo(ui_traits_data)

    browser_output_label = LabelWidget("From browser...")

    def browser_state_changed_cb(new_state):
        browser_output_label.set_text(f"From browser: {new_state.entityReferences()}")
```

```
    ui_request = NotebookHostUIRequest(stateChangedCallback=browser_state_changed_cb)

    initial_browser_state = ui_delegate.populateUI(
        ui_traits_data, UIAccess.kRead, ui_request, context
    )

    # Add the detached widget to the container.
    container.append(initial_browser_state.nativeData())
    container.append(browser_output_label)

    ######################################################################
    # Now move on to the inline entity box.

    container.append(LabelWidget("Type some text and press ENTER"))

    # UI traits for an inline entity box.
    ui_traits_data = TraitsData({EntityProviderTrait.kId, SingularTrait.kId, InlineTrait.kId})
    # We'll ask the UI delegate to mutate the container to add the
    # inline entity box.
    InPlaceTrait.imbueTo(ui_traits_data)

    inline_entity_output_label = LabelWidget("From inline...")

    def inline_entity_state_changed_cb(new_state):
        inline_entity_output_label.set_text(f"From inline: {new_state.entityReferences()}")

    ui_request = NotebookHostUIRequest(
        # Note that due to `InPlaceTrait`, we need to provide the
        # container widget within the request, so it can be populated.
        nativeData=container,
        stateChangedCallback=inline_entity_state_changed_cb,
    )

    ui_delegate.populateUI(ui_traits_data, UIAccess.kRead, ui_request, context)

    # Widget was added to the container by the delegate, so we only need
    # to add the host-side elements.
    container.append(inline_entity_output_label)
    container.attach()


closure()
```

second/entity

From browser: [EntityReference('notebook://second/entity')]

Type some text and press ENTER

some_entity

From inline: [EntityReference('notebook://some_entity')]

updateRequestCallback

A typcial workflow might be to browse for an asset, then populate an inline entity box with the result. The key functionality we need to do this is another communication channel - this time from the host application to the UI delegate. For this we'll make use of the `UIDelegateState` object's `updateRequestCallback`.

In [18]:

```
class NotebookUIDelegateState(NotebookUIDelegateState):
    """
    Redefine to add support for updateRequestCallback.
    """

    def __init__(self):
        super().__init__()
        self.__update_request_cb = None

    def updateRequestCallback(self):
        # Override base class, which just returns None.
        return self.__update_request_cb

    def setUpdateRequestCallback(self, update_request_cb):
        self.__update_request_cb = update_request_cb


class NotebookUIDelegateInterface(NotebookUIDelegateInterface):
    def populateUI(self, uiTraitsData, uiAccess, request, context, hostSession):

        if not SingularTrait.isImbuedTo(uiTraitsData):
            # We can only supply a single entity reference.
            return None

        current_state = NotebookUIDelegateState()
```

```
        def on_value_changed_cb(some_text):
            if request.stateChangedCallback() is None:
                return
            current_state.setEntityReferences([EntityReference("notebook://" + some_text)])
            request.stateChangedCallback()(current_state)

        ui_trait_set = uiTraitsData.traitSet()

        if {EntityProviderTrait.kId, InlineTrait.kId} <= ui_trait_set:
            widget = TextBoxWidget(on_value_changed_cb)

            # <===== New code here
            def on_request_updated(new_request):
                if new_request is None:
                    # None means "request finished", i.e. clean up.
                    return
                if entity_refs := new_request.entityReferences():
                    # Set the text box text to the stringified entity
                    # reference, without the "notebook://" prefix.
                    widget.set_text(entity_refs[0].toString()[len("notebook://"):])

            current_state.setUpdateRequestCallback(on_request_updated)
            on_request_updated(request)  # Initialise using initial request.
            # ==================>

        elif {EntityProviderTrait.kId, BrowserTrait.kId} <= ui_trait_set:
            widget = DropdownWidget(on_value_changed_cb, ["first/entity", "second/entity"])

        else:
            # We don't support other kinds of UI.
            return None

        if DetachedTrait.isImbuedTo(uiTraitsData):
            # Host wants a reference to the new widget.
            current_state.setNativeData(widget)

        if InPlaceTrait.isImbuedTo(uiTraitsData):
            # Host wants a container to be populated.
            container = request.nativeData()
            container.append(widget)

        return current_state
```

Note that we must check if the `new_request` is `None`. If so, it means the host is done with the request, i.e. the UI element is or should be destroyed and all callbacks and state are now invalid. The UI delegate should clean up in this case, e.g. disposing of any dangling references.

We can now link the two UI elements on the host side. We'll re-order widget creation, and make them both detached so we can assemble them in the container in the correct order.

In [19]:
```
class NotebookHostUIRequest(NotebookHostUIRequest):
    """
    Redefine to add support for entityReferences.
    """

    def __init__(self, entityReferences=None, **kwargs):
        super().__init__(**kwargs)
        self.__entity_references = entityReferences

    def entityReferences(self):
        # Override base class implementation, which just returns an
        # empty list.
        return self.__entity_references

    def setEntityReferences(self, entity_references):
        self.__entity_references = entity_references


def closure():
    ui_delegate = make_ui_delegate()

    # UI for inline entity box.

    ui_traits_data = TraitsData(
        {EntityProviderTrait.kId, SingularTrait.kId, InlineTrait.kId, DetachedTrait.kId}
    )

    inline_entity_output_label = LabelWidget("")

    def inline_entity_state_changed_cb(new_inline_entity_state):
        inline_entity_output_label.set_text(
            f"Result: {new_inline_entity_state.entityReferences()}"
        )

    inline_entity_ui_request = NotebookHostUIRequest(
        # Set default initial request state.
```

```
    entityReferences=[EntityReference("notebook://default/entity")],
    stateChangedCallback=inline_entity_state_changed_cb,
)

initial_inline_entity_state = ui_delegate.populateUI(
    ui_traits_data, UIAccess.kRead, inline_entity_ui_request, context
)

# UI for an entity browser.

ui_traits_data = TraitsData(
    {EntityProviderTrait.kId, SingularTrait.kId, BrowserTrait.kId, DetachedTrait.kId}
)

# When the browser provides a state update, update the request to
# the inline entity box with the entity references from the browser.
def browser_state_changed_cb(new_browser_state):
    if update_request_cb := initial_inline_entity_state.updateRequestCallback():
        inline_entity_ui_request.setEntityReferences(new_browser_state.entityReferences())
        update_request_cb(inline_entity_ui_request)

browser_ui_request = NotebookHostUIRequest(stateChangedCallback=browser_state_changed_cb)

initial_browser_state = ui_delegate.populateUI(
    ui_traits_data, UIAccess.kRead, browser_ui_request, context
)

# Assemble everything into a container and render.

    container = ContainerWidget()
    # Add the detached widget to the container.
    container.append(initial_browser_state.nativeData())
    container.append(LabelWidget("Choose from above, or type some text and press ENTER"))
    container.append(initial_inline_entity_state.nativeData())
    container.append(inline_entity_output_label)
    container.attach()


closure()
```

second/entity

Choose from above, or type some text and press ENTER

second/entity

Result: [EntityReference('notebook://second/entity')]

We now have two delegated widgets, both a black box as far as the host is concerned, but linked together via the host using the `stateChangedCallback` of the `UIDelegateRequest` and the `updateRequestCallback` of the `UIDelegateState`.

Note that we also initialised the `UIDelegateRequest` for the inline entity box with a default entity reference. Both the initial `UIDelegateRequest` provided to `populateUI`, and the initial `UIDelegateState` returned from `populateUI` should be filled in with as much information as available at the time. Then, both the `UIDelegateRequest` and the `UIDelegateState` can be updated dynamically via the `updateRequestCallback` and `stateChangedCallback`, respectively.

It is in this way that we set up two unidirectional communication channels between the host and the UI delegate, which together provide asynchronous communication back and forth.

entityTraitsDatas

So far we've made use of the `nativeData` and `entityReferences` properties of the `UIDelegateRequest` and `UIDelegateState`. The final property to cover is the `entityTraitsDatas` property of the `UIDelegateRequest` and `UIDelegateState`. This can be used in the `UIDelegateRequest` to provide entity metadata to the UI delegate, for example for use as a filter predicate. The UI delegate can populate the `entityTraitsDatas` of a `UIDelegateState` to communicate entity metadata back to the host, for example to augment the metadata of an entity to be published.

Let's turn to a publishing workflow. We will query the UI delegate for a browser for image entities to target, which will return a target entity reference along with some custom metadata that the host doesn't necessarily understand, but which the manager will appreciate when publishing.

For simplicity, we'll rewrite the `populateUI` method to only support this use-case. In practice an implementation would support our existing use-cases too, branching as appropriate for the given parameters.

In [20]:
```
from openassetio_mediacreation.specifications.application import WorkfileSpecification
from openassetio_mediacreation.specifications.twoDimensional import (
    BitmapImageResourceSpecification,
)
from openassetio_mediacreation.traits.ui import MetadataProviderTrait
```

```
# Our backend database of entities.
database = {
    "workfiles": {
        "project/workfile/first": {},
        "project/workfile/second": {},
    },
    "plates": {
        "project/plate/background": {"mimeType": "image/jpeg"},
        "project/plate/web": {"mimeType": "image/gif"},
        "project/plate/head": {"mimeType": "image/png"},
    },
}


class NotebookUIDelegateState(NotebookUIDelegateState):
    """
    Redefine to add support for entityTraitsDatas.
    """

    def __init__(self):
        super().__init__()
        self.__entity_traits_datas = []

    def entityTraitsDatas(self):
        # Override base class, which just returns an empty list.
        return self.__entity_traits_datas

    def setEntityTraitsDatas(self, entity_traits_datas):
        self.__entity_traits_datas = entity_traits_datas


class NotebookUIDelegateInterface(NotebookUIDelegateInterface):
    def populateUI(self, uiTraitsData, uiAccess, request, context, hostSession):
        if uiAccess != UIAccess.kWrite:
            # This UI delegate only supports write (publishing)
            # use-cases.
            return None

        ui_trait_set = uiTraitsData.traitSet()

        if not ({EntityProviderTrait.kId, SingularTrait.kId, BrowserTrait.kId,
                 DetachedTrait.kId}
                <= ui_trait_set):
            # This UI delegate only supports providing a detached entity
            # browser for a single entity.
            return None
        entity_traits_datas = request.entityTraitsDatas()
        if not entity_traits_datas:
            # We require traits to determine what entities should show
            # in the browser.
            return None

        # Determine which entity type the host wants to display.

        desired_entity_traits_data = entity_traits_datas[0]
        desired_entity_trait_set = desired_entity_traits_data.traitSet()

        # Check that the desired trait set is a superset (or equal to) a
        # supported trait set.
        if WorkfileSpecification.kTraitSet <= desired_entity_trait_set:
            entity_type = "workfiles"
        elif BitmapImageResourceSpecification.kTraitSet <= desired_entity_trait_set:
            entity_type = "plates"
        else:
            # We only support either workfiles (aka project files) or
            # images.
            return None

        entity_ids = list(database[entity_type].keys())

        if entity_type == "plates":
            # Further filter by MIME type, if provided.
            specification = BitmapImageResourceSpecification(desired_entity_traits_data)
            desired_mime_type = specification.locatableContentTrait().getMimeType()
            if desired_mime_type is not None:
                # We should support comma-separated MIME types, as well
                # as wildcards. See LocatableContentTrait for details.
                desired_mime_types = desired_mime_type.split(",")
                entity_ids = [
                    entity_id
                    for entity_id in entity_ids
                    if database["plates"][entity_id]["mimeType"] in desired_mime_types
                ]

        current_state = NotebookUIDelegateState()

        def on_value_changed_cb(entity_id):
```

```
            current_state.setEntityReferences(
                [EntityReference("notebook://" + entity_id + "/new")]
            )

            # Additional metadata, if the host asked for it.
            if MetadataProviderTrait.kId in ui_trait_set:
                custom_entity_traits = TraitsData()
                custom_entity_traits.setTraitProperty(
                    "openassetio:example.User", "login", "Bob")
                current_state.setEntityTraitsDatas([custom_entity_traits])

            if state_changed_cb := request.stateChangedCallback():
                state_changed_cb(current_state)

        widget = DropdownWidget(on_value_changed_cb, entity_ids)

        current_state.setNativeData(widget)

        return current_state
```

Switching to the host now, let's create an image asset browser, and display any metadata returned by the UI delegate.

In [21]:

```
class NotebookHostUIRequest(NotebookHostUIRequest):
    """
    Redefine to add support for entityTraitsDatas.
    """

    def __init__(self, entityTraitsDatas=None, **kwargs):
        super().__init__(**kwargs)
        self.__entity_traits_datas = entityTraitsDatas

    def entityTraitsDatas(self):
        # Override base class implementation, which just returns an
        # empty list.
        return self.__entity_traits_datas


def closure():
    ui_delegate = make_ui_delegate()

    ui_traits_data = TraitsData(
        {
            EntityProviderTrait.kId,
            MetadataProviderTrait.kId,
            SingularTrait.kId,
            BrowserTrait.kId,
            DetachedTrait.kId,
        }
    )

    image_specification = BitmapImageResourceSpecification.create()
    # We'll only support png and jpg images
    image_specification.locatableContentTrait().setMimeType("image/jpeg,image/png")

    entity_ref_and_traits_output = ipywidgets.Output()

    def browser_state_changed_cb(new_browser_state):
        result_txt = ""
        entity_refs = new_browser_state.entityReferences()
        if entity_refs:
            result_txt += f"Entity '{entity_refs[0]}'"
            entity_traits = new_browser_state.entityTraitsDatas()
            if entity_traits:
                result_txt += " with traits: "
                result_txt += str(entity_traits[0])

        entity_ref_and_traits_output.clear_output()
        with entity_ref_and_traits_output:
            print(result_txt)

    browser_ui_request = NotebookHostUIRequest(
        entityTraitsDatas=[image_specification.traitsData()],
        stateChangedCallback=browser_state_changed_cb,
    )

    initial_browser_state = ui_delegate.populateUI(
        ui_traits_data, UIAccess.kWrite, browser_ui_request, context
    )

    initial_browser_state.nativeData().attach()
    display(entity_ref_and_traits_output)


closure()
```

project/plate/head

```
Entity 'notebook://project/plate/head/new' with traits: {'openassetio:example.User': {'login': 'Bob'}}
```

Note that our "asset browser" (dropdown) only contains image entities, and only those with a supported MIME type. In this case, the additional `TraitsData` could be merged with the entity's traits during publishing, to provide additional (opaque) metadata that the asset management system cares about, but which has no meaning to the host application.

There are a couple of subtly introduced features of note here. Firstly, note we're adding `MetadataProviderTrait` (as well as `EntityProviderTrait`). The `MetadataProviderTrait` indicates that the host wants the `entityTraitsDatas` property to be populated by the UI delegate.

Secondly, we've changed our access mode from `UIAccess.kRead` to `UIAccess.kWrite`. The access mode is used in much the same way as in manager plugins (e.g. `ResolveAccess` for `resolve()`, `PublishingAccess` for `register()`, etc). The access mode is used to indicate the kind of subsequent operation that the host intends to perform on the result, i.e. whether the provided entity references/traits will be used by the host for reading existing data, or for publishing new data to. For example, this can be used to determine whether to display an asset browser equivalent of "File->Open" (`kRead`) vs. "File->Save As" (`kWrite`) dialog.

uiPolicy

In our previous example, the UI delegate was very selective in the types of UI element that were supported (only a detached entity browser widget for `kWrite` usage). Filling in all the details required for a `populateUI` call might be an expensive operation for a host application, only for it to return `None`. In addition, there may be other sundry metadata that's needed ahead of time, for the host to properly adapt its native UI to the needs of the UI delegate.

This is where the `uiPolicy` method comes in. This method mirrors the `managementPolicy` method of manager plugins. We can use it to filter out early which kinds of UI the UI delegate plugin will support, as well as retrieve other policy-specific metadata.

Continuing with the publishing browser example, let's implement its `uiPolicy`

In [22]:
```python
from openassetio_mediacreation.traits.uiPolicy import ManagedTrait


class NotebookUIDelegateInterface(NotebookUIDelegateInterface):
    def uiPolicy(self, uiTraits, uiAccess, context, hostSession):
        policy = TraitsData()

        if uiAccess != UIAccess.kWrite:
            return policy

        if (
                not {EntityProviderTrait.kId, SingularTrait.kId, BrowserTrait.kId,
                     DetachedTrait.kId}
                    <= uiTraits
        ):
            return policy

        managed_trait = ManagedTrait(policy)
        managed_trait.imbue()  # Not strictly necessary -`setExclusive`, below, will do this implicitly.
        # We want the host to hide any native UI in favour of delegated
        # UI.
        managed_trait.setExclusive(True)

        return policy
```

The `uiPolicy` method should always return a `TraitsData`. The host can then check if the `ManagedTrait` is imbued, in order to determine whether a `populateUI` query may succeed.

The `get`/`setExclusive` property on the `ManagedTrait` provides additional metadata, instructing the host to hide its native alternative to the delegated UI. For example, the host could display a tabbed browser, one tab for files and one (delegated) for assets. If the exclusive property is true, then the files tab should be hidden.

Other policy metadata may be provided. For example the `DisplayNameTrait` may be imbued with a name for the UI element, to be used as a tab title.

Note that one condition `populateUI` checks for is not replicated in `uiPolicy` - the `entityTraitsDatas` trait set check. This illustrates why it is not sufficient to assume `populateUI` will always succeed, based solely on the `uiPolicy`. There may be cases, based on a more fine-grained analysis, that `populateUI` is not supported for a given set of arguments.

For completeness, let's see how this is used on the host side.

In [23]:
```python
ui_delegate = make_ui_delegate()

detached_entity_browser_for_read_policy = ui_delegate.uiPolicy(
    {EntityProviderTrait.kId, SingularTrait.kId, BrowserTrait.kId, DetachedTrait.kId},
    UIAccess.kRead,
    context,
)

inplace_entity_browser_for_publish_policy = ui_delegate.uiPolicy(
```

```
    {EntityProviderTrait.kId, SingularTrait.kId, BrowserTrait.kId, InPlaceTrait.kId},
    UIAccess.kWrite,
    context,
)

detached_entity_browser_for_publish_policy = ui_delegate.uiPolicy(
    {EntityProviderTrait.kId, SingularTrait.kId, BrowserTrait.kId, DetachedTrait.kId},
    UIAccess.kWrite,
    context,
)

print(ManagedTrait(detached_entity_browser_for_read_policy).isImbued())
print(ManagedTrait(detached_entity_browser_for_read_policy).getExclusive())
print(ManagedTrait(inplace_entity_browser_for_publish_policy).isImbued())
print(ManagedTrait(inplace_entity_browser_for_publish_policy).getExclusive())
print(ManagedTrait(detached_entity_browser_for_publish_policy).isImbued())
print(ManagedTrait(detached_entity_browser_for_publish_policy).getExclusive())
```

```
False
None
False
None
True
True
```

We also note that both `uiPolicy` and `populateUI` are parametrised by additional arguments - `context` and `hostSession`. These too can be introspected to determine appropriate behaviour and support, in a similar way to manager plugins. For example, the `context.locale` property can be examined to determine broader traits of the current logical process, and the `hostSession.host().identifier()` could be used to branch based on the specific host application that the plugin is loaded into.

Disposing of UI elements

So far we've assumed that UI elements live forever. However, it is common for UI elements to be destroyed and re-created, e.g. for one-off dialogs. If a UI delegate holds on to state that outlives the UI element, then we run the risk of memory leaks, or worse, fatal errors when the host disposes of a UI element but does not dispose of the UI delegate's connection to it.

It may be that all references are rooted in the UI element, so that when that is destroyed, so is all dangling state (e.g. in callback closures). Or it may be that the host application and its UI framework has a way to notify listeners that a UI element is defunct, and this native mechanism can be used to ensure proper cleanup.

However, we cannot rely on these assumptions. OpenAssetIO UI delegates provide two mechanisms to handle this case.

- When a single delegated UI element is disposed, the host can pass `None` (or `nullptr` in C++) to the `updateRequestCallback`, signalling that the request is finished and the UI delegate should clear all its references and destroy any internal state related to the particular UI element.
- When the UI delegate as a whole is no longer valid, along with all delegated UI elements associated with it, then the `close()` method can be called. This method is called automatically on destruction of the `UIDelegate` class, but it can also be called by hosts directly in order to reset and reuse a UI delegate. Host applications should call `close()` (or destroy the UI delegate) whenever they are wholesale destroying all UI associated with the UI delegate.

The particular responsibilities of the host and the UI delegate with respect to clean-up (e.g. which is responsible for destroying the UI element itself) will vary between applications and their UI frameworks, and any ambiguities must be documented by the host application.

Let's see this in action in a plugin that provides an inline entity reference box. We'll need to make the plugin stateful in order to track active widget states, so (rather contrived) we'll create a closure around the widget request, and persist it in a container.

In [24]:
```
class NotebookUIDelegateInterface(NotebookUIDelegateInterface):
    def __init__(self):
        super().__init__()
        self.__memos = set()

    # <===== Important code here
    def close(self, hostSession):
        self.__memos.clear()

    # ========================>

    def populateUI(self, uiTraitsData, uiAccess, request, context, hostSession):
        if uiAccess != UIAccess.kRead:
            return None
        if not ({EntityProviderTrait.kId, SingularTrait.kId, InlineTrait.kId,
                 DetachedTrait.kId}
                <= uiTraitsData.traitSet()):
            return None

        current_state = NotebookUIDelegateState()

        def memo():
```

```
            def on_value_changed_cb(some_text):
                if state_changed_cb := request.stateChangedCallback():
                    current_state.setEntityReferences([EntityReference("notebook://" + some_text)])
                    state_changed_cb(current_state)

            def on_request_updated(new_request):
                # <===== Important code here
                if new_request is None:
                    self.__memos.remove(memo)
                    return
                # ========================>
                current_state.setEntityReferences(new_request.entityReferences())

            widget = TextBoxWidget(on_value_change_cb=on_value_changed_cb)
            current_state.setNativeData(widget)

            current_state.setUpdateRequestCallback(on_request_updated)

            self.__memos.add(memo)
            return current_state

        return memo()
```

First lets see what happens when we naively clean up on the host side, but don't clean up the UI delegate

In [25]:
```
import weakref
import gc


ui_traits_data = TraitsData(
    {EntityProviderTrait.kId, SingularTrait.kId, InlineTrait.kId, DetachedTrait.kId}
)


def closure():
    ui_delegate = make_ui_delegate()
    ui_request = NotebookHostUIRequest(stateChangedCallback=lambda state: None)
    ui_state = ui_delegate.populateUI(ui_traits_data, UIAccess.kRead, ui_request, context)
    weakref_to_ui_request = weakref.ref(ui_request)

    del ui_request
    del ui_state
    gc.collect()

    print(f"Request cleaned up? {weakref_to_ui_request() is None}")


closure()
```

```
Request cleaned up? False
```

We see that the UI request is leaked. So let's explicitly clean up the references by passing `None` to the `updateRequestCallback`.

In [26]:
```
def closure():
    ui_delegate = make_ui_delegate()
    ui_request = NotebookHostUIRequest(stateChangedCallback=lambda state: None)
    ui_state = ui_delegate.populateUI(ui_traits_data, UIAccess.kRead, ui_request, context)
    weakref_to_ui_request = weakref.ref(ui_request)

    ui_state.updateRequestCallback()(None)

    del ui_request
    del ui_state
    gc.collect()

    print(f"Request cleaned up? {weakref_to_ui_request() is None}")


closure()
```

```
Request cleaned up? True
```

This did the job. However, if there are many UI elements associated with a UI delegate, finding them all and calling the `updateRequestCallback` could be arduous.

To clean up everything in one go we can call the `close()` method of the UI delegate associated with the widget(s).

In [27]:
```
def closure():
    ui_delegate = make_ui_delegate()
    ui_request = NotebookHostUIRequest(stateChangedCallback=lambda state: None)
    ui_state = ui_delegate.populateUI(ui_traits_data, UIAccess.kRead, ui_request, context)
    weakref_to_ui_request = weakref.ref(ui_request)

    ui_delegate.close()

    del ui_request
```

```
    del ui_state
    gc.collect()

    print(f"Request cleaned up? {weakref_to_ui_request() is None}")


closure()
```

```
Request cleaned up? True
```

The `close()` method is also called automatically on destruction of the UI delegate instance. So equivalently to the above, we can destroy the UI delegate itself.

In [28]:
```
def closure():
    ui_delegate = make_ui_delegate()
    ui_request = NotebookHostUIRequest(stateChangedCallback=lambda state: None)
    ui_state = ui_delegate.populateUI(ui_traits_data, UIAccess.kRead, ui_request, context)
    weakref_to_ui_request = weakref.ref(ui_request)

    del ui_delegate

    del ui_request
    del ui_state
    gc.collect()

    print(f"Request cleaned up? {weakref_to_ui_request() is None}")


closure()
```

```
Request cleaned up? True
```

Sharing data between manager and UI delegate plugins

The data used by the UI delegates in the examples above (for example, the entities in the "asset browser" dropdown) was all hardcoded. In reality, UI delegate plugins will usually need to consult some external source of information. It is up to the plugin how this data is retrieved. Typically, though, the UI delegate plugin may wish to (re)use functionality available in the corresponding manager plugin. There is no direct link between the `UIDelegateInterface` instance and the `ManagerInterface` instance in the API. However, there *is* a link via a `Context` object, constructed by the manager plugin, and passed along with every call to the `populateUI` method (for example).

The `ManagerInterface` has an opportunity to furnish a `Context` with an arbitrary `managerState` object on construction, which can then be accessed by the UI delegate plugin.

To see how this might work, we'll create an entity browser (dropdown) with data populated from state communicated via the `Context`.

First we'll need to define a state object that inherits from `ManagerStateBase` (this is crucial, since it allows transport of Python state through the C++ middleware). Then we'll need to implement the required methods in the manager plugin, by augmenting our `NotebookManagerInterface`, to make use of this state object.

In [29]:
```
from openassetio.managerApi import ManagerStateBase


# The backend database.
backend_database = ["first/entity", "second/entity", "third/entity"]


class NotebookManagerState(ManagerStateBase):

    def __init__(self, database):
        ManagerStateBase.__init__(self)
        self.database = database


class NotebookManagerInterface(NotebookManagerInterface):
    def hasCapability(self, capability):
        """
        Context won't be constructed with managerState unless we
        advertise that we're capable of providing it.
        """
        return (
                super().hasCapability(capability)
                or capability == ManagerInterface.Capability.kStatefulContexts
        )

    def createState(self, hostSession):
        """
        This will be called whenever the host creates a Context object
        and the result stored on Context.managerState.
        """
        return NotebookManagerState(backend_database)

    def createChildState(self, parentState, hostSession):
        """
```

```
        This will be called whenever the host creates a child Context
        object, used to subgroup together logical units of work.

        We re-use the same database from the parent
        """
        return NotebookManagerState(parentState.database)
```

Now we'll assume the `context` provided to the UI delegate has this state available

In [30]:
```
class NotebookUIDelegateInterface(NotebookUIDelegateInterface):
    def populateUI(self, uiTraitsData, uiAccess, request, context, hostSession):

        if (
                not {DetachedTrait.kId, SingularTrait.kId, EntityProviderTrait.kId,
                     BrowserTrait.kId}
                    <= uiTraitsData.traitSet()
        ):
            return None

        current_state = NotebookUIDelegateState()

        def on_value_changed_cb(some_text):
            if request.stateChangedCallback() is None:
                return
            current_state.setEntityReferences([EntityReference("notebook://" + some_text)])
            request.stateChangedCallback()(current_state)

        widget = DropdownWidget(
            on_value_changed_cb,
            # <===== Important code here
            context.managerState.database,
            # ========================>
        )

        current_state.setNativeData(widget)

        return current_state
```

Now on the host application side, we'll need to load our updated manager and UI delegate plugins, get a `Context` from the manager, and pass it in to the UI delegate's `populateUI` method.

In [31]:
```
def closure():
    manager = make_manager()
    # Context now contains a custom `managerState`.
    context = manager.createContext()

    ui_delegate = make_ui_delegate()

    browser_output = LabelWidget("From browser...")

    def state_changed_cb(new_state):
        browser_output.set_text(f"From browser: {new_state.entityReferences()}")

    ui_traits_data = TraitsData(
        {EntityProviderTrait.kId, SingularTrait.kId, BrowserTrait.kId, DetachedTrait.kId}
    )
    ui_request = NotebookHostUIRequest(stateChangedCallback=state_changed_cb)

    initial_browser_state = ui_delegate.populateUI(
        ui_traits_data, UIAccess.kRead, ui_request, context
    )

    initial_browser_state.nativeData().attach()
    browser_output.attach()


closure()
```

From browser: [EntityReference('notebook://third/entity')]

We've successfully communicated state from the manager plugin to the UI delegate via the OpenAssetIO `Context`.

C++ specifics

Usage in C++ is, as with most of OpenAssetIO, much the same as usage in Python. There are the usual translations needed (e.g. `None` becomes a `shared_ptr` holding a `nullptr`), but also one or two C++ specifics.

nativeData

In Python, the `nativeData` held by `UIDelegateRequestInterface` and `UIDelegateStateInterface` classes can be any arbitrary object - trivial to do in Python. But of course C++ is strongly typed, so we can't return arbitrary objects from function calls. To work around this the `nativeData()` method in C++ returns a `std::any`, which must be unpacked via `std::any_cast` to the actual native type.

Python -> C++ nativeData

When the host and UI delegate plugin are written in different languages, specifically Python and C++, the usage of `nativeData` is a special case.

To determine when this case applies at runtime, we can make use of the `info()` method. For host applications, there is an `info()` method on the `UIDelegate` instance. For plugins, there is an `info()` method on the `HostInterface` instance (accessible via `hostSession.host().info()`). This returns an arbitrary dictionary, and, as alluded to earlier, some of the keys in this dictionary have a special meaning. In particular the `openassetio.constants.kInfoKey_IsPython` (`"isPython"`) entry can be `True` to indicate that the plugin/host is written in Python. The default Python base class implementation of `info()` on both the `HostInterface` and `UIDelegateInterface` adds this key to the dictionary.

When Python provides C++ with a `nativeData`, then the `std::any` returned by `nativeData()` will always contain a CPython `PyObject*`. This is a non-owning reference to the Python `nativeData` object - in particular, the host should *not* attempt to decrement the reference count of this object (unless incrementing it first).

It is up to the C++ code to further unpack the `PyObject*`, if necessary. For example, Qt `QWidget` widgets can be converted to/from `PyObject` using Qt's Shiboken API.

Conversely, when C++ provides a `nativeData` to Python, the `std::any` returned from `nativeData()` *must* contain a `PyObject*`. The Python bindings will raise an exception if this is not the case when the `nativeData()` method is called in Python.

It is then up to the C++ code to properly clean up the `PyObject*` that it placed in the `std::any` when it is no longer needed (e.g. decrement the reference count on destruction of the `std::any`, if required).

Context.managerState

As mentioned, the `ManagerStateBase` base class is required to allow the `managerState` object to pass through C++ and Python unmodified. Internally, it's (ab)using a technicality of the Python/C++ bindings and C++ `virtual` inheritance. The `ManagerStateBase` class itself is an empty C++ class, bound to Python.

This means you can inherit from `ManagerStateBase` in C++/Python and store an instance in the `Context`, and this instance will persist as it travels into Python/C++ and back out. However, the instance will only be accessible in the language in which the subclass was defined. That is, a C++ `Context.managerState` will appear empty in Python, and vice versa. This causes complications for hybrid C++/Python plugins, and/or where one of the manager or UI delegate plugins is written in C++ and the other is written in Python.

It is currently an exercise to the plugin author to decide how best to work around this case. The `openassetio::python::converter` namespace's `castToPyObject` might be of some help here.