Anvil Labs

The Anvil Labs project team

May 02, 2023

CONTENTS

CHAPTER

ONE

MODULES

1.1 Atomic

Object-oriented state management. The aim is to separate the state from the UI. Atoms manage the state. A form now has two jobs. It displays the state and hooks up events (like button clicks) to actions. The atom's job is to manage the state. Each atom is a global object, which any form can import. By eliminating state from forms, there is no need to pass state up and down the form hierarchy. It also makes testing easier, since we can test atoms in isolation.

1.1.1 Examples

Some examples can be found at in this clone link.

Counter

```
# Create an atom that holds state
from anvil_labs.atomic import atom, action, selector

@atom
class CountAtom:
    value = 0

    @selector
    def get_count(self):
        return self.value

    @action
    def update_count(self, increment):
        self.value += increment

count_atom = CountAtom()
```

```
# Create a form to display the count
from anvil_labs.atomic import render
from ..atoms.count import count_atom

class Count(CountTemplate):
    def __init__(self):
        self.display_count()
```

(continues on next page)

(continued from previous page)

```
    @render
    def display_count(self):
        # I get called any time the get_count return value changes
        self.count_lbl.text = count_atom.get_count()

    def neg_btn_click(self, **event_args):
        count_atom.update_count(-1)

    def pos_btn_click(self, **event_args):
        count_atom.update_count(1)
```

In this example, whenever a button is clicked:

- the button event handler calls an `action` on the atom,
- which updates the state of the atom,
- which then updates any `selectors` that depend on that state change; and finally,
- any `render` methods that depend on those updates are re-rendered.

Action → State change → Re-compute selectors → Call render methods

1.1.2 Terminology

Action

An action is an expression/statement that updates the state of an atom. Whenever the state changes, the atomic module invokes a render cycle. We probably don't want each state change to invoke a render cycle; sometimes it makes sense to combine state updates into a single action. To combine actions into a single action, use the `@action` decorator. Using the `@action` decorator means that the render cycle will only be invoked after all actions within the decorated function have been completed. The `@action` decorator can be used on any function and does not necessarily need to be a method of an atom.

Selector

A `selector` is a method of an atom that returns a value based on the atom's state. Essentially it is a getter method. When a `selector` needs to do something expensive, by say, combining various attributes of an atom, use the `@selector` decorator. The return value from a decorated selector method is cached. Whenever the atoms state changes, if the selector depends on that state, the selector's return value will be re-computed.

You should never update the state (call an action) within a `selector`.

Render

A `render` is any method/function that depends on the state of an atom, or that depends on the return value of a `selector`.

It's most commonly used on methods within forms, but a `render` can be used outside of a form.

```
from anvil.js.window import document
from anvil_labs.atomic import render

@render
def update_tab_title():
    document.title = count_atom.get_count()

update_tab_title()
```

Note we might want to do this with the `autorun` function. The above example is equivalent to.

```
from anvil_labs.atomic import autorun

def update_tab_title():
    document.title = count_atom.get_count()

autorun(update_tab_title)
```

To depend on the state of an atom, the `render` method must explicitly access that state.

```
# BAD Example
class Count(CountTemplate):
    def __init__(self):
        self.display_count(count_atom.value)

    @render
    def display_count(self, count):
        self.count_lbl.text = count
```

In the above example, the `display_count` method does not explicitly access the `count_atom.value` attribute. This means it does **not** depend on this attribute. The code should look like this:

```
# GOOD Example
class Count(CountTemplate):
    def __init__(self):
        self.display_count()

    @render
    def display_count(self):
        self.count_lbl.text = count_atom.value
```

Accessing an attribute/selector implicitly subscribes the `render` method to changes in the state of those attributes/selectors. Any time one of these attributes changes, the `render` method is invoked (re-rendered).

You should never update the state (call an action) within a `render` method.

If the render method is called by a component, it will only execute when the form is on the screen. This prevents renders from happening for cached forms, or forms that are no longer active.

Atom

An `atom` is any object that knows how to register subscribers and request renders. To create an atom, use the `@atom` decorator.

Whenever an attribute of an `atom` is a `list` or `dict`, the attribute will be converted to a `ListAtom` or `DictAtom`. Each is a subclass of `list`/ `dict` and behave as you'd expect. The only difference is that these classes know how to register subscribers and request renders when their state changes.

Subscriber

A subscriber is an advanced feature. It's the final part of the render cycle. After all renders have been completed any subscribers that were decorated with the `@subscribe` decorator will be called. A subscriber takes a single argument, a tuple of actions that were called to invoke the render cycle.

A reason to use a subscriber might be to update storage based on an action that was invoked.

Here's an example.

```
from anvil_extras.storage import indexed_db
from anvil_labs.atomic import atom, action, subscribe

@atom
class Todos:
    def __init__(self):
        self.todos = indexed_db.get("todos", [])

    @action(update_db=True)
    def add_todo(self, todo):
        self.todos = self.todos + [todo]


todos_atom = Todos()

@subscribe
def update_db_subscriber(actions):
    if any(hasattr(action, "update_db") for action in actions):
        indexed_db["todos"] = todos_atom.todos
```

The `@action` decorator can be used on any function or method. If the decorator is used above a method then the `atom` used as the `self` argument can be caught within a `subscribe` function

```
@subscribe
def update_db_subscriber(actions):
    for action in actions:
        if hasattr(action, "update_db"):
            atom = action.atom
            break
    else:
        return

    # now use the atom to do something specific
    ...
```

Alternative Approaches to the subscriber

```
# ALTERNATIVE APPROACH 1 - use a render

is_first_run = True
@render
def update_db_with_render():
    global is_first_run

    todos = [dict(todo) for todo in todos_atom.todos]
    # accessing the todos and each converting each todo to a dict
    # creates a dependency on the todos and each key of each todo
    # whenever these change this method is called

    if is_first_run:
        is_first_run = False
        return
    indexed_db["todos"] = todos

update_db_with_render()

# ALTERNATIVE APPROACH 2 - use autorun

def update_db_with_render():
    # same code as above
    ...
autorun(update_db_with_render)


# ALTERNATIVE APPROACH 3 - use a reaction

def update_db_with_reaction(todos):
    indexed_db["todos"] = todos

reaction(lambda: [dict(todo) for todo in todos_atom.todos], update_db_with_reaction)
# the first function sets up the dependencies
# the return value of this function is passed to the reaction function
# the reaction function is called only after the first change to any dependency
```

1.1.3 Bindings and Writeback

It's not recommended to use anvil writebacks and data bindings with atoms. This is because we can't control the render cycle.

Instead, there are two helper functions to create bindings and writebacks in code.

```
from anvil_labs.atomic import bind

class Count(CountTemplate):
    def __init__(self):
        bind(self.count_lbl, "text", count_atom.get_count)
        # or bind it to an attribute of an atom
        bind(self.count_lbl, "text", count_atom, "value")
```

The bind method is equivalent to:

```
def bind(component, prop, atom_or_selector, attr=None):
    @render(bound=component)
    def render_bind():
        if callable(atom_or_selector):
            setattr(component, prop, atom_or_selector())
        elif isinstance(atom_or_selector, dict):
            setattr(component, prop, atom_or_selector[attr])
        else:
            setattr(component, prop, getattr(atom_or_selector, attr))

    render_bind()
```

Note the render decorator can take a bound parameter. This means that the render won't fire if the component is not on the screen. This is not necessary when using the render decorator on a form method.

A writeback is similar to a bind, but a list of events must be provided.

```
writeback(self.check_box, "checked", self.item, "completed", events=["change"])
```

Alternatively, the writeback can be called with a selector in place of the atom and an action in place of the atom attribute. If the selector/action call signature is used, the action must take a single argument (the updated property of the component).

```
writeback(component, prop, atom, attr, events)
writeback(component, prop, selector, action, events)
```

1.1.4 API

set_debug(*debug=False*)

Show logging output for the module

@atom

Create an atom class. An atom class knows how to register subscribers and request re-renders when its state changes.

@portable_atom

Create an atom class which is also a portable class. It is recommended to use the `@portable_atom` decorator over a combination of `@atom` and `@portable_class`.

@render

@render(*bound=None*)

Use the `render` decorator anytime you want a function to depend on atom attributes or selectors. In a `render` method the attributes must be accessed explicitly. Whenever one of the attributes of the atom changes, the `render` method will be invoked.

@action

@action(***kws*)

The action decorator should be used above any method that you want to combine actions into a single action. A base action changes the state of an atom. When calling a function decorated with the `@action` decorator, the render cycle will be invoked only after all actions within the function have been executed. It's worth noting that the decorator doesn't need to be used unless you want to combine state updates into a single action. In the counter example, the action decorator is unnecessary, since there is only a single state update within the function (updating the `.value` property)

`@selector`

The selector decorator can only be used on methods within an atom. Its utility is caching the return value and a selector subscribes to atom attributes in a similar way to renders. If any attribute changes, the cached value will be re-computed. It's worth noting that the selector decorator is unnecessary on methods where accessing the attribute is cheap. In the counter example, the selector is unnecessary and adds little to the implementation.

`autorun`(*fn*)

`autorun`(*fn*, *bound=None*)

Similar to `render(fn)()`. Any atom attributes accessed within the body of the function will trigger a new call to the function when changed.

Calling `autorun` returns a dispose function. When the dispose function is called it stops any future renders of this `autorun` function.

`autorun` can be used as a decorator - but note that the returned function is not the original function but the dispose function.

`reaction`(*depends_on_fn*, *then_react_fn*, *, *fire_immediately=False*, *include_previous=False*)

a `reaction` is similar to a `render`. Changes in the `depends_on_fn` will force the `then_react_fn` to be called. The `depends_on_fn` is a function that takes no args. It should access any attributes that, when changed, should result in the call to the `then_react_fn`.

If `depends_on_fn` returns a value that is not `None`, this value will be passed to the `then_react_fn`. If you need the previous result returned from `depends_on_fn` set `include_previous=True`. If `incldue_previous` is `True` then the call signature for `then_react_fn` should take 2 argments, the current return value and the previous return value from the `depnds_on_fn`.

`depends_on_fn` will fire immediately. But the `then_react_fn` is only called the next time a dependency changes. To call the `then_react_fn` immediately set `fire_immediately=True`.

It would be rare to need to use this function.

However, in cases where you want to react to a change in an atom's state that may result in a subsequent change, in another atom's state a reaction may be useful. It can also be used as an alternative to `autorun` or `render`.

See the example above for alternative approaches to updating `indexed_db`

The reaction method returns a dispose function that can be called when you want to stop reactions.

`@subscribe`

A subscriber is called after all re-renders resulting from a series of actions a subscriber takes a single argument - the tuple of actions that caused the re-render. See examples for use cases.

`unsubscribe`(*f*)

Stop a subscriber from running.

`class DictAtom`

A subclass of `dict`. Any attribute within an atom that is a `dict` will be converted to a `DictAtom`. This allows render methods to depend on keys of dicts within the atom's state.

`class ListAtom`

A subclass of `list`. Any attribute within an atom that is a `list` will be converted to a `ListAtom`. Renders that depend on the `ListAtom` will only be invoked if the `ListAtom` changes through methods like `remove()`, `clear()` etc.

`class Atom`(***kws*)

A portable atom class that can be called with kwargs. Each kwarg will become an attribute of the atom. Useful if you prefer to access attributes rather than keys of a `DictAtom`.

e.g. `todo_atom = Atom(done=False, description='walk the dog')`

`ignore_updates`

This can be used as a context manager (using `with`) to update an atom without invoking a render cycle. A reason to use this decorator is to lazy load an atom property. Use with caution.

1.1.5 Gotchas and advanced concepts

My component isn't updating

Make sure that you have used the render decorator and that you have called this method from the `__init__` function.

Why don't you use `self.init_components(**properties)` in the example?

The primary job of `init_components` is to set up data bindings. But since we don't have any data bindings, we don't need to use this method. Note that `init_components` does more work when used within a custom component.

How do I lazy load an attribute?

You can use the `ignore_updates` decorator to prevent actions invoking render cycles. And since calling an action within a render or selector is not allowed it becomes necessary.

```
import anvil.server
from anvil_labs.atomic import atom, ignore_updates, selector

@atom
class Todos:
    def __init__(self):
        self._todos = None

    @property
    @selector
    def todos(self):
        if self._todos is None:
            with ignore_updates:
                self._todos = anvil.server.call("get_todos")
        return self._todos
```

Alternatively, you can call an action, ensuring that the action is not called inside a render/selector

```
from atoms import todos_atom

class Form1(Form1Template):
    def __init__(self, **properties):
        # fetch_todos is an action that calls the server if it needs to
        todos_atom.fetch_todos()
        self.display_todos()
```

My UI is taking a long time to update

That might be because you are calling a server function within an action. The fetch example is a good example of how to update the UI while you make a call.

```
@atom
class Fetch:
    value = None
    loading = False

    @action
    def set_status(self, value, loading=False):
        self.value = value
        self.loading = loading

    def do_fetch(self):
        self.set_status(None, loading=True)
        ret = anvil.server.call_s("do_fetch")
        self.set_status(ret, loading=False)

    @selector
    def get_info(self):
        return self.value, self.loading


fetch_atom = Fetch()
```

`do_fetch` is not an action, but `set_status` is an action. `set_status` is cheap and so the UI updates quickly. Each call to `set_status` invokes a render cycle. When `loading` is `True` the UI can disable a button while we call the server function.

How do I work with anvil data tables?

We're working on it.

1.2 Fido

This module provides an interface for FIDO WebAuthn integration in Anvil applications. It enables device registration and authentication using FIDO devices, such as security keys or biometric authenticators.

1.2.1 Demo App

Clone Link
Live Demo

1.2.2 Requirements

webauthn <https://pypi.org/project/webauthn/> must be installed on the server.

1.2.3 Functions

`register_device()`

This function registers a FIDO device for the currently logged-in user. The user table must have a 'fido' simple object column.

`login_with_fido(email: str)`

This function attempts to authenticate the user with the provided email address using their registered FIDO device. The email might be stored in IndexedDB, local storage, or a similar client-side storage system.

param email
The email address of the user attempting to authenticate.

returns
The authenticated user object, or None if authentication fails.

1.2.4 Internal Functions

`generate_registration()`

This function generates a registration request for a FIDO device.

returns
A public key for device registration.

`verify_registration(response)`

This function verifies the registration response for a FIDO device.

param response
The response from the FIDO device registration process.

returns
The result of the verification process.

`generate_authentication_options(email)`

This function generates authentication options for a FIDO device.

param email
The email address of the user attempting to authenticate.

returns
The authentication options for the FIDO device.

`verify_authentication_options(authentication_options)`

This function verifies the authentication options for a FIDO device.

param authentication_options
The authentication options for the FIDO device.

returns
The result of the verification process, or None if an error occurs.

1.3 Kompot

Kompot exposes a serialization mechanism for Python builtins and Anvil portable classes.

Kompot provides wrappers for `anvil.server.call()` and `@anvil.server.callable` to take advantage of the enhanced serialization between client and server calls.

Wikipedia on kompot:

> In 1885, Lucyna Ćwierczakiewiczowa wrote in a recipe book that **kompot** preserved fruit so well it seemed fresh

1.3.1 Builtins

In addition to standard JSONable types, kompot supports the following builtins:

`set`, `frozenset`, `tuple`, `date`, `datetime`

`dict` objects will preserve their order and keys can be arbitrary.

`type` objects registered with kompot can also be serialized.

1.3.2 API

`register`(*cls*)

All portable classes that kompot can serialize must be registered by calling `register(cls)`.

```
from anvil_labs import kompot
import anvil.server

@anvil.server.portable_class
class Foo:
    ...

kompot.register(Foo)
```

(kompot.register can also be used as a decorator)

`serialize`(*obj*)

Serialize an arbitrary object into a JSONable object.

If kompot does not know how to handle the object, it will be left untouched.

Kompot does not know how to handle objects like:

- anvil table rows
- media objects
- capabilities

We leave Anvil to serialize these objects when calling the server.

`preserve`(*obj*)

Like `serialize` but will throw a `SerializationError` if there are any unhandled objects.

Use `preserve` for storing an object as a simple object.

Use `serialize` for sending an object from the client to the server.

reconstruct(*obj*)

Reconstruct an object from the output of `serialize` or `preserve`

call(*fn_name*, **args*, ***kws*)

call_s(*fn_name*, **args*, ***kws*)

call_async(*fn_name*, **args*, ***kws*)

Use inplace of `anvil.server.call()`

Kompot will serialize the args and kws and reconstruct the returned value.

The server function must be decorated with `@kompot.callable`.

@callable

Use inplace of `@anvil.server.callable`.

Kompot will reconstruct the serialized args and kws, call the original function and then serialize the return value.

Must be combined with `kompot.call()`.

batch_call()

A context manager for batching calls

```
from anvil_labs import kompot

with kompot.batch_call(silent=True) as c:
    c.call("foo", x=1)
    c.call("bar", 42)

foo_result, bar_result = c.result
```

1.4 NonBlocking

Call function in a non-blocking way.

1.4.1 Examples

Call a server function

After making updates on the client, call a server function to update the database. In this example, we don't care about the return.

```
from anvil_labs.non_blocking import call_async

def button_click(self, **event_args):
    self.update_database()
    self.open_form("Form1")

def update_database(self):
    # Unlike anvil.server.call we do not wait for the call to return
    call_async("update", self.item)
```

If you care about the return value, you can provide handlers.

```
from anvil_labs.non_blocking import call_async

def handle_result(self, res):
    print(res)
    Notification("successfully saved").show()

def handle_error(self, err):
    print(err)
    Notification("there was a problem", style="danger").show()

def update_database(self, **event_args):
    call_async("update", self.item).on_result(self.handle_result, self.handle_error)
    # Equivalent to
    async_call = call_async("update", self.item)
    async_call.on_result(self.handle_result, self.handle_error)
    # Equivalent to
    async_call = call_async("update", self.item)
    async_call.on_result(self.handle_result)
    async_call.on_error(self.handle_error)
```

repeat

Call a function repeatedly using the `repeat()` function. After each interval seconds the function will be called. To end or cancel the repeated call use the `cancel` method.

```
from anvil_labs import non_blocking

i = 0
def do_heartbeat():
    global heartbeat, i
    if i >= 42:
        heartbeat.cancel()
        # equivalent to non_blocking.cancel(heartbeat)
    print("da dum")
    i += 1

heartbeat = non_blocking.repeat(do_heartbeat, 1)
```

defer

Call a function after a set period of time using the `defer()` function. To `cancel` the deferred call, use the `cancel()` method.

```
from anvil_labs import non_blocking

pending = []

def do_save():
    global pending
    pending, saves = [], pending
    if not saves:
```

(continues on next page)

(continued from previous page)

```
        return
    anvil.server.call_s("save", saves)

deferred_save = None

def on_save(saves):
    global pending, deferred_save
    non_blocking.cancel(deferred_save)
    # we could also use deferred_save.cancel() but we start with None
    pending.extend(saves)
    deferred_save = non_blocking.defer(do_save, 1)

# calling on_save() repeatedly will cancel the current do_save deferred call and create␣
↪a new one
```

1.4.2 API

call_async(*fn*, **args*, ***kws*)

call_async(*fn_name*, **args*, ***kws*)

Returns an `AsyncCall` object. The *fn* will be called in a non-blocking way.

If the first argument is a string then the server function with name *fn_name* will be called in a non-blocking way.

wait_for(*async_call_object*)

Blocks until the `AsyncCall` object has finished executing.

class AsyncCall

Don't call this directly, instead use the above functions.

on_result(*self*, *result_handler*, *error_handler=None*)

Provide a result handler to handle the return value of the non-blocking call. Provide an optional error handler to handle the error if the non-blocking call raises an exception. Both handlers should take a single argument.

Returns `self`.

on_error(*self*, *error_handler*)

Provide an error handler that will be called if the non-blocking call raises an exception. The handler should take a single argument, the exception to handle.

Returns `self`.

await_result(*self*)

Waits for the non-blocking call to finish executing and returns the result. Or raises an exception if the non-blocking call raised an exception.

property result

If the non-blocking call has not yet completed, raise a `RuntimeError`.

If the non_blocking call has completed returns the result. Or raises an exception if the non-blocking call raised an exception.

property error

If the non-blocking call has not yet completed, raise a `RuntimeError`.

If the non-blocking call raised an exception the exception raised can be accessed using the `error` property. The error will be `None` if the non-blocking call returned a result.

property set_status

One of `"PENDING"`, `"FULFILLED"`, `"REJECTED"`

cancel(*ref*)

Cancel an active call to `delay` or `defer`. The first argument should be `None` or the the return value from a call to `delay` or `defer`.

Calling `cancel(ref)` is equivalent to `ref.cancel()`. You may wish to use `cancel(ref)` if you start with a placeholder `ref` equal to `None`. See the `defer` example above.

repeat(*fn*, *interval*)

Repeatedly call a function with a set interval (in seconds)

`fn` should be a callable that takes no args. `interval` should be an `int` or `float` representing the time in seconds between function calls.

The function is called in a non-blocking way.

A call to `repeat` returns a `RepeatRef` object that has a `.cancel()` method.

Calling the `.cancel()` method will stop the next call repeated call from executing.

defer(*fn*, *delay*)

Defer a function call after a set period of time has elapsed (in seconds).

`fn` should be a callable that takes no args. `delay` should be an `int` or `float` representing the time in seconds.

The function is called in a non-blocking way. A call to `defer` returns a `DeferRef` object that has a `.cancel()` method.

Calling the `.cancel()` method will stop the deferred function from executing.

1.5 WebWorker

The `web_worker` module supports creating client-side background tasks.

It is a limited API and should only be used for computational heavy calcuations on the client. If you're doing something on the client that seems to be making the page unresponsive, then a web worker might be for you.

You will need to include the following script tags in your Native Libraries for this module to work:

```
<script src="_/theme/anvil-labs/worker.js" defer></script>
```

1.5.1 Example

Create a worker module - say `fib_worker`

```
def fib(num):
    a, b = 1, 0
    i = 0
    print(worker) # worker is a global object injected into worker module
    while i < num:
        a, b = a + b, a
        i += 1
        worker.task_state["i"] = i
    return b
```

In your code use the `web_worker` module call the `fib` function as a background client side task

```
from anvil_labs.web_worker import Worker

my_worker = Worker("fib_worker")

class Form1(Form1Template):
    def fib_result(self, result):
        alert(result)

    def fib_error(self, error):
        raise error

    def timer_tick(self, state):
        if self.task.is_completed():
            self.timer.interval = 0
        else:
            print(self.task.get_state().get("i"))

    def button_1_click(self, **event_args):
        self.task = my_worker.launch_task("fib", 2**20)
        self.timer.interval = 1
        my_worker.on_result(self.fib_result)
        my_worker.on_error(self.fib_error)
        # my_worker.on_state_change(self.fib_state_change)
```

1.5.2 Notes

A worker module, like `fib_worker` above, can only import libraries from python's standard lib.

Exceptions raised should be from the standard lib.

Only JSONable objects can be passed to and from the worker to the client.

A Worker object can only launch a single background task at any one time. You can create more Worker objects if you want multiple tasks to run in parallel.

The API for client side `web_worker` matches Anvil's API for launching and communicating with background tasks. Using a Timer to check the current state of the web worker task will work in the same way.

There are 3 additions to the api - `task.on_result()`, `task.on_error()`, `task.on_state_change()`. You should pass a callback to these methods that can be used to receive information from the task object.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

P

R

S

U

V

W