Anvil Extras

Owen Campbell, Stuart Cork

Mar 11, 2021

CONTENTS

CHAPTER

ONE

INSTALLATION

Anvil-Extras is intended to be used as a dependency for other Anvil applications.

1. Clone anvil-extras to your account:

2. Add anvil-extras as a dependency to your own app(s):
   - From the gear icon at the top of your app's left hand sidebar, select 'Dependencies'
   - From the 'Add a dependency' dropdown, select 'Anvil-Extras'

That's it! You should now see the extra components available in your app's toolbox on the right hand side and all the other features are available for you to import.

CHAPTER

TWO

CONTRIBUTING

All contributions to this project are welcome via pull request (PR) on the Github repository

2.1 Issues

Please open an Issue and describe the contribution you'd like to make before submitting any code. This prevents duplication of effort and makes reviewing the eventual PR much easier for the maintainers.

If you could start the name of the branch you work on with the number of the issue you create, that would be very helpful as github will automatically link the two together.

2.2 Commits

Please try to use commit messages that give a meaningful history for anyone using git's log features. Try to use messages that complete the sentence, "This commit will…" There is some excellent guidance on the subject from Chris Beams

Please ensure that your commits do not include changes to either *anvil.yaml* or *.anvil_editor.yaml*.

2.3 Components

All the components in the library are intended to work from the anvil toolbox as soon as the dependency has been added to an application, without any further setup. This means that they cannot use any of the features within the library's theme.

If you are thinking of submitting a new component, please ensure that it is entirely standalone and does not require any css or javascript from within a theme element or native library.

2.4 Python Code

- Please try, as far as possible, to follow PEP8
- Use the Black formatter to format all code and the isort utility to sort import statements.

2.5 Documentation

Please include documentation for your contribution as part of your PR. Our documents are written in reStructuredText and hosted at Read The Docs

Our docs are built using Sphinx which you can install locally and use to view your work before submission. To build a local copy of the docs in a 'build' directory:

```
sphinx-build docs build
```

You can then open 'index.html' from within the build directory using your favourite browser.

2.6 Testing

The project uses the Pytest library and its test suite can be run with:

```
python -m pytest
```

We appreciate the difficulty of writing unit tests for Anvil applications but, if you are submitting pure Python code with no dependency on any of the Anvil framework, we'll expect to see some additions to the test suite for that code.

2.7 Merging

We require both maintainers to have reviewed and accepted a PR before it is merged.

If you would like feedback on your contribution before it's ready to merge, please create a draft PR and request a review.

2.8 Copyright

By submitting a PR, you agree that your work may be distributed under the terms of the project's licence and that you will become one of the project's joint copyright holders.

CHAPTER

THREE

MESSAGING

3.1 Introduction

This library provides a mechanism for forms (and other components) within an Anvil app to communicate in a 'fire and forget' manner.

It's an alternative to raising and handling events - instead you 'publish' messages to a channel and, from anywhere else, you subscribe to that channel and process those messages as required.

3.2 Usage

3.2.1 Create the Publisher

You will need to create an instance of the Publisher class somewhere in your application that is loaded at startup.

For example, you might create a client module at the top level of your app called 'common' with the following content:

```
from .messaging import Publisher

publisher = Publisher()
```

and then import that module in your app's startup module/form.

3.2.2 Publish Messages

From anywhere in your app, you can import the publisher and publish messages to a channel. e.g. Let's create a simple form that publishes a 'hello world' message when it's initiated:

```
from ._anvil_designer import MyPublishingFormTemplate
from .common import publisher


class MyPublishingForm(MyPublishingFormTemplate):

    def __init__(self, **properties):
        publisher.publish(channel="general", title="Hello world")
        self.init_components(**properties)
```

The publish method also has an optional 'content' parameter which can be passed any object.

3.2.3 Subscribe to a Channel

Also, from anywhere in your app, you can subscribe to a channel on the publisher by providing a handler function to process the incoming messages.

The handler will be passed a Message object, which has the title and content of the message as attributes.

e.g. On a separate form, let's subscribe to the 'general' channel and print any 'Hello world' messages:

```
from ._anvil_designer import MySubscribingFormTemplate
from .common import publisher


class MySubscribingForm(MySubscribingFormTemplate):

    def __init__(self, **properties):
        publisher.subscribe(
            channel="general", subscriber=self, handler=self.general_messages_handler
        )
        self.init_components(**properties)

    def general_messages_handler(self, message):
        if message.title = "Hello world":
            print(message.title)
```

You can unsubscribe from a channel using the publisher's *unsubscribe* method.

You can also remove an entire channel using the publisher's *close_channel* method.

Be sure to do one of these if you remove instances of a form as the publisher will hold references to those instances and the handlers will continue to be called.

3.2.4 Logging

By default, the publisher will log each message it receieves to your app's logs (and the output pane if you're in the IDE).

You can change this default behaviour when you first create your publisher instance:

```
from .messaging import Publisher
publisher = Publisher(with_logging=False)
)
```

The *publish*, *subscribe*, *unsubscribe* and *close_channel* methods each take an optional *with_logging* parameter which can be used to override the default behaviour.

CHAPTER

FOUR

NAVIGATION

A client module for that provides dynamic menu construction.

4.1 Introduction

This module builds a menu of link objects based on a simple dictionary definition.

Rather than manually adding links and their associated click event handlers, the module does that for you!

4.2 Usage

4.2.1 Forms

In order for a form to act as a target of a menu link, it has to register a name with the navigation module using a decorator on its class definition. e.g. Assuming the module is installed as a dependency named 'Extras':

```
from ._anvil_designer import HomeTemplate
from anvil import *
from anvil_extras import navigation


@navigation.register(name="home")
class Home(HomeTemplate):
  def __init__(self, **properties):
    self.init_components(**properties)
```

4.2.2 Menu

- In the Main form for your app, add a content panel to the menu on the left hand side and call it 'menu_panel'
- Add a menu definition dict to the code for your Main form and pass the panel and the dict to the menu builder. e.g.

```
from ._anvil_designer import MainTemplate
from anvil import *
from anvil_extras import navigation
from HashRouting import routing

menu = [
```

(continues on next page)

(continued from previous page)

```
  {"text": "Home", "target": "home"},
  {"text": "About", "target": "about"},
]


class Main(MainTemplate):

  def __init__(self, **properties):
    self.advanced_mode = False
    navigation.build_menu(self.menu_panel, menu)
    self.init_components(**properties)
```

will add 'Home' and 'About' links to the menu which will open registered forms named 'home' and 'about' respectively.

Each item in the dict needs the 'text' and 'target' keys as a minimum. It may also include 'full_width', 'routing' and 'visibility' keys:

- 'full_width' can be True or False to indicate whether the target form should be opened with 'full_width_row' or not.
- 'routing' can be either 'classic' or 'hash' to indicate whether clicking the link should use Anvil's *add_component* function or hash routing to open the target form. Classic routing is the default if the key is not present in the menu dict.
- 'visibility' can be a dict mapping an anvil event to either True or False to indicate whether the link should be made visible when that event is raised.

All other keys in the menu dict are passed to the Link constructor.

For example, to add icons to each of the examples above, a 'Contact' item that uses hash routing and a 'Settings' item that should only be visible when advanced mode is enabled:

```
from ._anvil_designer import MainTemplate
from anvil import *
from Navigation import navigation
from HashRouting import routing

menu = [
  {"text": "Home", "target": "home", "icon": "fa:home"},
  {"text": "About", "routing": "hash", "target": "about", "icon": "fa:info"},
  {"text": "Contact", "routing": "hash", "target": "contact", "icon": "fa:envelope"},
  {
    "text": "Settings",
    "target": "settings",
    "icon": "fa:gear",
    "visibility": {
      "x-advanced-mode-enabled": True,
      "x-advanced-mode-disabled": False
    }
  }
]


@routing.main_router
class Main(MainTemplate):

  def __init__(self, **properties):
```

(continues on next page)

(continued from previous page)

```
    self.advanced_mode = False
    navigation.build_menu(self.menu_panel, menu)
    self.init_components(**properties)

  def form_show(self, **event_args):
    self.set_advanced_mode(False)
```

Note - since this example includes hash routing, it also requires a decorator from the [Hash Routing App](https://github.com/s-cork/HashRouting) on the Main class.

4.2.3 Startup

In order for the registration to occur, the form classes need to be loaded before the menu is constructed. This can be achieved by using a startup module and importing each of the forms in the code for that module.

e.g. Create a module called 'startup', set it as the startup module and import your Home form before opening the Main form:

```
from anvil import open_form
from .Main import Main
from . import Home

open_form(Main())
```

4.2.4 Page Titles

By default, the menu builder will also add a Label to the title slot of your Main form. If you register a form with a title as well as a name, the module will update that label as you navigate around your app. e.g. to add a title to the home page example:

```
from ._anvil_designer import HomeTemplate
from anvil import *
from anvil_extras import navigation


@navigation.register(name="home", title="Home")
class Home(HomeTemplate):
  def __init__(self, **properties):
    self.init_components(**properties)
```

If you want to disable this feature, set the *with_title* argument to *False* when you call *build_menu* in your Main form. e.g.

```
class Main(MainTemplate):

  def __init__(self, **properties):
    self.advanced_mode = False
    navigation.build_menu(self.menu_column_panel, menu, with_title=False)
    self.init_components(**properties)
```

CHAPTER

FIVE

AUTHORISATION

A server module that provides user authentication and role based authorisation for server functions.

5.1 Installation

You will need to setup the Users and Data Table services in your app:

- Ensure that you have added the 'Users' service to your app
- **In the 'Data Tables' service, add:**
  - a table named 'permissions' with a text column named 'name'
  - a table named 'roles' with a text column named 'name' and a 'link to table'

  column named 'permissions' that links to multiple rows of the permissions table * a new 'link to table' column in the Users table named 'roles' that links to multiple rows of the 'roles' table

5.2 Usage

5.2.1 Users and Permissions

- Add entries to the permissions table. (e.g. 'can_view_stuff', 'can_edit_sensitive_thing')
- Add entries to the roles table (e.g. 'admin') with links to the relevant permissions
- In the Users table, link users to the relevant roles

5.2.2 Server Functions

The module includes two decorators which you can use on your server functions:

authentication_required

Checks that a user is logged in to your app before the function is called and raises an error if not. e.g.:

```
import anvil.server
from anvil_extras.authorisation import authentication_required

@anvil.server.callable
@authentication_required
```

(continues on next page)

(continued from previous page)

```
def sensitive_server_function():
  do_stuff()
```

authorisation_required

Checks that a user is logged in to your app and has sufficient permissions before the function is called and raises an error if not:

```
import anvil.server
from anvil_extras.authorisation import authorisation_required

@anvil.server.callable
@authorisation_required("can_edit_sensitive_thing")
def sensitive_server_function():
  do_stuff()
```

You can pass either a single string or a list of strings to the decorator. The function will only be called if the logged in user has ALL the permissions listed.

Notes:

- The import lines in the examples above assume you have installed the Extras ibrary as a

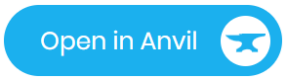
dependency. If you used direct inclusion, you will need to import from your own copy of the module. * The order of the decorators matters. *anvil.server.callable* must come before either of the authorisation module decorators.

CHAPTER

SIX

AUGMENTATION

A client module for adding custom jQuery events to any anvil component



6.1 Examples

```
from anvil_extras import augment
augment.set_event_handler(self.link, 'hover', self.link_hover)
# equivalent to
# augment.set_event_handler(self.link, 'mouseenter', self.link_hover)
# augment.set_event_handler(self.link, 'mouseleave', self.link_hover)
# or
# augment.set_event_handler(self.link, 'mouseenter mouseleave', self.link_hover)

def link_hover(self, **event_args):
  if 'enter' in event_args['event_type']:
    self.link.text = 'hover'
  else:
    self.link.text = 'hover_out'

#================================================
# augment.set_event_handler equivalent to
augment.add_event(self.button, 'focus')
self.button.set_event_handler('focus', self.button_focus)

def button_focus(self, **event_args):
 self.button.text = 'Focus'
 self.button.role =  'secondary-color'
```

6.2 need a trigger method?

```
# 'augment' the component by adding any event...
augment.add_event(self.textbox, 'select')
# augment.add_event(self.textbox, 'custom')
# augment.add_event(self.textbox, '')

def button_click(self, **event_args):
  self.textbox.trigger('select')
```

6.3 Keydown example

```
augment.set_event_handler(self.text_box, 'keydown', self.text_box_keydown)

def text_box_keydown(self, **event_args):
  key_code = event_args.get('key_code')
  key = event_args.get('key')
  if key_code == 13:
    print(key, key_code)
```

6.4 advanced feature

you can prevent default behaviour of an event by returning a value in the event handler function - example use case*

```
augment.set_event_handler(self.text_area, 'keydown', self.text_area_keydown)

def text_area_keydown(self, **event_args):
  key = event_args.get('key')
  if key.lower() == 'enter':
    # prevent the standard enter new line behaviour
    # prevent default
    return True
```

CHAPTER

SEVEN

POPOVER

A client module that allows bootstrap popovers in anvil

Live Example: popover-example.anvil.app

Example Clone Link:

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search