Anvil Extras

The Anvil Extras project team

Nov 29, 2021

CONTENTS

CHAPTER

ONE

INSTALLATION

There are two options for installling anvil-extras:

1. As a third-party dependency

   This is the simplest option. After you add the library to your app, there is no further maintenance involved and updates will happen automatically.

2. As a clone

   This option involves using git on your local machine to manage your own copy of the anvil-extras library. There is more work involved but you gain full control over when and if your copy is updated.

NOTE: If you are an enterprise user, you cannot use the third-party dependency option.

1.1 Install as a third-party dependency

- From the gear icon at the top of your app's left hand sidebar, select 'Dependencies'
- In the buttons to the right of 'Add a dependency', click the 'Third Party' button
- Enter the id of the Anvil-Extras app: C6ZZPAPN4YYF5NVJ
- Hit enter and ensure that the library appears in your list of dependencies
- Select whether you wish to use the 'Development' or 'Published' version

For the published version, the dependency will be automatically updated as new versions are released. On the development version, the update will occur whenever we merge new changes into the library's code base.

Whilst we wouldn't intentionally merge broken code into the development version, you should consider it unstable and not suitable for production use.

1.2 Install as a clone

1.2.1 Clone the Repository

- In your browser, navigate to your blank Anvil Extras app within your Anvil IDE.
- From the App Menu (with the gear icon), select 'Version History…' and click the 'Clone with Git' button.
- Copy the displayed command to you clipboard.
- In your terminal, navigate to a folder where you would like to create your local copy
- Paste the command from your clipboard into your terminal and run it.

- You should now have a new folder named 'Anvil_Extras'.

1.2.2 Configure the Remote Repositories

Your local repository is now configured with a known remote repository pointing to your copy of the app at Anvil. That remote is currently named 'origin'. We will now rename it to something more meaningful and also add a second remote pointing to the repository on github.

- In your terminal, navigate to your 'Anvil_Extras' folder.
- Rename the 'origin' remote to 'anvil' with the command:

```
git remote rename origin anvil
```

- Add the github repository with the command:

```
git remote add github git@github.com:anvilistas/anvil-extras.git
```

1.2.3 Update your local app

To update your app, we will now fetch the latest version from github to your local copy and push it from there to Anvil.

- In your terminal, fetch the lastest code from github using the commands:

```
git fetch github
git reset --hard github/main
```

- Finally, push those changes to your copy of the app at Anvil:

```
git push -f anvil
```

1.2.4 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.

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.

If your component has custom properties or events, it must be able to cope with multiple instances of itself on the same form. There are examples of how to do this using a unique id in several of the existing components.

Whilst canvas based components will be considered, the preference is for solutions using standard Anvil components, custom HTML forms and css.

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.

Add the licence text and copyright statement to the top of your code.

Ensure that there is a line with the current version number towards the top of your code.

This can be automated by using pre-commit. To use `pre-commit`, first install `pre-commit` with pip and then run `pre-commit install` inside your local `anvil-extras` repository. All commits thereafter will be adjusted according to the above `anvil-extras` python requirements.

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

COMPONENTS

3.1 Autocomplete

A material design TextBox with autocomplete. A subclass of TextBox - other properties, events and methods inherited from TextBox.

3.1.1 Properties

suggestions list[str]

A list of autocomplete suggestions

suggest_if_empty bool

If True then autocomplete will show all options when the textbox is empty

3.1.2 Events

suggestion_clicked When a suggestion is clicked. If a suggestion is selected with enter the `pressed_enter` event fires instead.

3.2 Chip

A variation on a label that includes a close icon. Largely based on the Material design Chip component.

3.2.1 Properties

text str

Displayed text

icon icon

Can be a font awesome icon or a media object

close_icon boolean

Whether to include the close icon or not

foreground color the color of the text and icons

background color background color for the chip

spacing_above str

One of `"none"`, `"small"`, `"medium"`, `"large"`

spacing_below str

One of `"none"`, `"small"`, `"medium"`, `"large"`

visible bool

Is the component visible

3.2.2 Events

close_click When the close icon is clicked

click When the chip is clicked

show When the component is shown

hide When the component is hidden

3.3 ChipsInput

A component for adding tags/chips. Uses a Chip with no icon.

3.3.1 Properties

chips tuple[str]

the text of each chip displayed. Empty strings will be ignored, as will duplicates.

primary_placeholder str

The placeholder when no chips are displayed

secondary_placeholder str

The placeholder when at least one chip is displayed

spacing_above str

One of `"none"`, `"small"`, `"medium"`, `"large"`

spacing_below str

One of `"none"`, `"small"`, `"medium"`, `"large"`

visible bool

Is the component visible

3.3.2 Events

chips_changed When a chip is added or removed

chip_added When a chip is added. Includes the chip text that was added as an event arg.

chip_removed When a chip is removed. Includes the chip text that was removed as an event arg;

show When the component is shown

hide When the component is hidden

3.4 Determinate ProgressBar

A linear progress bar displaying completion towards a known target.

3.4.1 Properties

track_colour Color

The colour of the background track

indicator_colour Color

The colour of the progress indicator bar

progress Number

Between 0 and 1 to indicate progress

3.5 EditableCard

A card to display a value and allow it to be edited by clicking.

3.5.1 Properties

editable Boolean

Whether the card should allow its value to be edited

icon Icon

To display in the top right corner of the card

datatype String

“text”, “number”, “date”, “time” or “yesno” Setting this property will affect which type of component is displayed to edit the value

3.6 Indeterminate ProgressBar

A linear progress bar to indicate processing of unknown duration.

3.6.1 Properties

track_colour Color

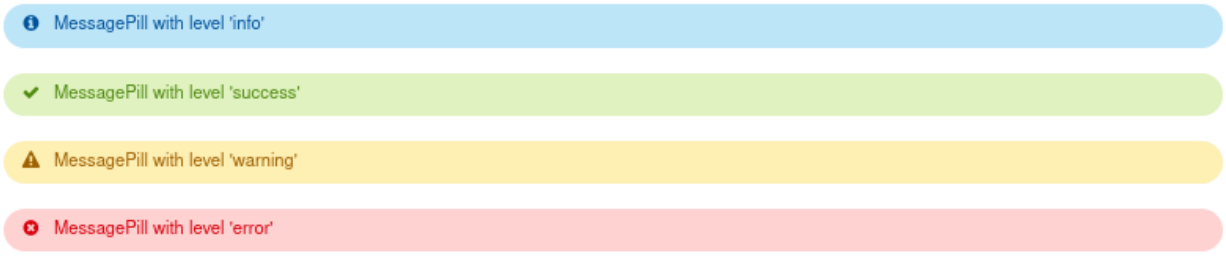
The colour of the background track

indicator_colour Color

The colour of the progress indicator bar

3.7 MessagePill

A rounded text label with background colour and icon in one of four levels.



3.7.1 Properties

level string

"info", "success", "warning" or "error"

message string

The text to be displayed

3.8 MultiSelectDropdown

A multi select dropdown component with optional search bar

3.8.1 Properties

align String

"left", "right", "center" or "full"

items Iterable of Strings, Tuples or Dicts

Strings and tuples as per Anvil's native dropdown component. More control can be added by setting the items to a list of dictionaries. e.g.

```
self.multi_select_drop_down.items = [
    {"key": "1st": "value": 1, "subtext": "pick me"},
    {"key": "2nd": "value": 2, "enabled": False},
    "---",
    {"key": "item 3": "value": 3, "title": "3rd times a charm"},
]
```

The "key" property is what is displayed in the dropdown. The value property is what is returned from the selected_values.

The remainder of the properties are optional.

"enabled" determines if the option is enabled or not - defaults to True.

"title" determines what is displayed in the selected box - if not set it will use the value from "key".

"subtext" adds subtext to the dropdown display.

To create a divider include "---" at the appropriate index.

placeholder String

Placeholder when no items have been selected

enable_filtering Boolean

Allow searching of items by key

multiple Boolean

Can also be set to false to disable multiselect

enabled Boolean

Disable interactivity

visible Boolean

Is the component visible

spacing_above String

One of "none", "small", "medium", "large"

spacing_below String

One of "none", "small", "medium", "large"

selected Object

get or set the current selected values.

3.8.2 Events

change When the selection changes

show When the component is shown

hide When the component is hidden

3.9 PageBreak

For use in forms which are rendered to PDF to indicate that a page break is required.

The optional `margin_top` property changes the amount of white space at the top of the page. You can set the `margin_top` property to a positive/negative number to adjust the whitespace. Most of the time this is unnecessary. This won't have any effect on the designer, only the generated PDF.

The optional `border` property defines the style of the component in the IDE. The value of the property affects how a `PageBreak` component looks in the browser during the execution. It has no effect in the generated PDF, where the component is never visible or in the IDE, where the component is always `"1px solid grey"`.

It is possible to change the default style for all the `PageBreak`s in the app by adding the following code to `theme.css`:

```
.break-container {
    border: 2px dashed red !important;
}
```

Using this technique rather than the `border` property affects how the component looks both in the IDE and at runtime.

3.10 Pivot

A pivot table component based on https://github.com/nicolaskruchten/pivottable

3.10.1 Properties

items list of dicts

The dataset to be pivoted

rows list of strings

attribute names to prepopulate in rows area

columns list of strings

attribute names to prepopulate in columns area

values list of strings

attribute names to prepopulate in vals area (gets passed to aggregator generating function)

aggregator string

aggregator to prepopulate in dropdown (e.g. "Count" or "Sum")

3.11 Quill Editor

A wrapper around the Quill editor.

3.11.1 Properties

auto_expand Boolean

When set to `True` the Editor will expand with the text. If `False` the height is the starting height.

content Object

This returns a list of dicts. The content of any Quill editor is represented as a Delta object. A Delta object is a wrapper around a JSON object that describes the state of the Quill editor. This property exposes the undelrying JSON which can then be stored in a data table simple object cell.

When you do `self.quill.content = some_object`, this will call the underlying `setContents()` method.

You can also set the `content` property to a string. This will call the underlying `setText()` method.

Retrieving the `content` property will always return the underlying JSON object that represents the contents of the Quill editor. It is equivalent to `self.quill.getContents().ops`.

enabled Boolean

Disable interactivity

height String

With auto_expand this becomes the starting height. Without auto_expand this becomes the fixed height.

modules Object

Additional modules can be set at runtime. See Quill docs for examples. If a toolbar option is defined in modules this will override the toolbar property.

placeholder String

Placeholder when there is no text

readonly Boolean

Check the Quill docs.

spacing_above String

One of `"none"`, `"small"`, `"medium"`, `"large"`

spacing_below String

One of `"none"`, `"small"`, `"medium"`, `"large"`

theme String

Quill supports `"snow"` or `"bubble"` theme.

toolbar Boolean or Object

Check the Quill docs. If you want to use an Object you can set this at runtime. See quill docs for examples.

visible Boolean

Is the component visible

3.11.2 Methods

All the methods from the Quill docs should work. You can use camel case or snake case. For example `self.quill.get_text()` or `self.quill.getText()`. These will not come up in the autocomplete.

Methods from the Quill docs call the underlying javascript Quill editor and the arguments/return values will be as described in the Quill documentation.

There are two Anvil specific methods:

get_html Returns a string representing the html of the contents of the Quill editor. Useful for presenting the text in a RichText component under the `"restricted_html"` format.

set_html Set the contents of the Quill editor to html. The html will be sanitized in the same way that a RichText component sanitizes the html. See Anvil's documentation on the RichText component.

3.11.3 Events

text_change When the text changes

selection_change When the selection changes

show When the component is shown

hide When the component is hidden

3.12 Slider

Slider component based on the Javascript library noUiSlider.

3.12.1 Properties

start number | list[number]

The initial values of the slider. This property determines the number of handles. It is a required property. In the designer use comma separated values which will be parsed as JSON.

connect "upper" | "lower" | bool | list[bool]

The connect option can be used to control the bar color between the handles or the edges of the slider. When using one handle, set the value to either `'lower'` or `'upper'` (equivalently `[True, False]` or `[False, True]`). For sliders with 2 or more handles, pass a list of True, False values. One value per gap. A single value of `True` will result in a coloured bar between all handles.

min number

Lower bound. This is a required property

max number

Upper bound. This is a required property

range object

An object with `'min'`, `'max'` as keys. For additional options see noUiSlider documentation. This does not need to be set and will be inferred from the `min`, `max` values.

step number

By default, the slider slides fluently. In order to make the handles jump between intervals, the step option can be used.

format Provide a format for the values. This can either be a string to call with .format or a format spec. e.g. `"{:.2f}"` or just `".2f"`. See python"s format string syntax for more options.

For a mapping of values to descriptions, e.g. `{1: 'strongly disagree', 2: 'agree', ...}` use a custom formatter. This is a dictionary object with `'to'` and `'from'` as keys and can be set at runtime. The `'to'` function takes a float or int and returns a str. The `'from'` takes a str and returns a float or int. See the anvil-extras Demo for an example.

value number

returns the value of the first handle. This can only be set after initialization or with a databinding.

values list[numbers]

returns a list of numerical values. One value for each handle. This can only be set after initialization or with a databinding.

formatted_value str

returns the value of the first handle as a formatted string, based on the format property

formatted_values list[str]

returns the a list of values as formatted strings, based on the format property

padding number | list[number, number]

Padding limits how close to the slider edges handles can be. Either a single number for both edges. Or a list of two numbers, one for each edge.

margin number

When using two handles, the minimum distance between the handles can be set using the margin option. The margin value is relative to the value set in `range`.

limit number

The limit option is the opposite of the margin option, limiting the maximum distance between two handles

animate bool

Set the animate option to False to prevent the slider from animating to a new value with when setting values in code.

behaviour str

This option accepts a `"-"` separated list of `"drag"`, `"tap"`, `"fixed"`, `"snap"`, `"unconstrained"` or `"none"`

tooltips bool

Adds tooltips to the sliders. Uses the same formatting as the format property.

pips bool

Sets whether the slider has pips (ticks).

pips_mode str

One of `'range'`, `'steps'`, `'positions'`, `'count'`, `'values'`

pips_values list[number]

a list of values. Interpreted differently depending on the mode

pips_density int

Controls how many pips are placed. With the default value of 1, there is one pip per percent. For a value of 2, a pip is placed for every 2 percent. A value of zero will place more than one pip per percentage. A value of -1 will remove all intermediate pips.

pips_stepped bool

the stepped option can be set to true to match the pips to the slider steps

color str

The color of the bars. Can be set to theme colors like `'theme:Primary 500'` or hex values `'#2196F3'`.

enabled bool

Disable interactivity

visible bool

Is the component visible

spacing_above str

One of `"none"`, `"small"`, `"medium"`, `"large"`

spacing_below str

One of `"none"`, `"small"`, `"medium"`, `"large"`

3.12.2 Methods

reset Resets the slider to its initial position i.e. it's `start` property

3.12.3 Events

slide Raised whenever the slider is sliding. The handle is provided as an argument to determine which handle is sliding.

change Raised whenever the slider has finished sliding. The handle is provided as an argument to determine which handle is sliding. Change is the writeback event.

show Raised when the component is shown.

hide Raised when the component is hidden.

3.13 Switch

A material design switch. A subclass of CheckBox.

3.13.1 Properties

checked boolean

checked_color Color

The background colour of the switch when it is checked

3.13.2 Events

changed Raised whenever the switch is clicked

3.14 Tabs

A simple way to implement tabs. Works well above another container abover or below. Set the container spacing property to none. It also understand the role material design role `'card'`

3.14.1 Properties

tab_titles list[str]

The titles of each tab.

active_tab_index int

Which tab should be active.

foreground color the color of the highlight and text. Defaults to `"theme:Primary 500"`

background color the background for all tabs. Defaults to `"transparent"`

role set the role to `'card'` or create your own role

align str

`"left"`, `"right"`, `"center"` or `"full"`

bold bool

applied to all tabs

italic bool

applied to all tabs

font_size int

applied to all tabs

font str

applied to all tabs

visible Boolean

Is the component visible

spacing_above String

One of `"none"`, `"small"`, `"medium"`, `"large"`

spacing_below String

One of `"none"`, `"small"`, `"medium"`, `"large"`

3.14.2 Events

tab_click When any tab is clicked. Inclues the paramters `tab_index` `tab_title` and `tab_component` as part of the `event_args`

show When the component is shown

hide When the component is hidden

CHAPTER

FOUR

MODULES

4.1 Animation

A wrapper around the Web Animations API

4.1.1 Interfaces

class Animation(*component*, *effect*)

An Animation object will be returned from the `Effect.animate()` method and the *`animate()`* function. Provides playback control for an animation.

class Effect(*transiton*, ***effect_timing_options*)

A combination of a *`Transition`* object and timing options. An effect can be used to animate an Anvil Component with its `.animate()` method. `effect_timing_options` are equivalent to those listed at EffectTiming The `effect_timing_options` have identical defaults to those listed at MDN, except `duration`, which defaults to `333ms`.

class Transition(***css_frames*)

A dictionary-based class. Each key should be a CSS/ transform property in camelCase with a list of frames. Each frame in the list represents a style to hit during the animation. The first value in the list is where the animation starts and the final value is where the animation ends. See *Pre-computed Transitions* for examples.

Unlike the Web Animations API the `transform` CSS property can be written as separate properties.

e.g. `transform=["translateX(0) scale(0)", "translateX(100%) scale(1)"]` becomes `Transform(scale=[0, 1], translateX=[0, "100%"])`.

A limitation of this approach is that all transform based properties must have the same number of frames.

The Web Animations API uses a keyframes object in place of the anvil_extras Transition object. A keyframes object is typically a dictionary of lists or list of dictionaries. Any `transition` argument in the `anvil_extras.animate` module can be replaced with a keyframes object. i.e. if you find an animation example on the web you can use its keyframes object directly without having to convert it to a *`Transition`* object.

animate(*component*, *transition*, ***timing_options*)

A shortcut for animating an Anvil Component. Returns an Animation instance.

4.1.2 Examples

Animate on show

Use the show event to animate an Anvil Component. This could could also be at the end of an `__init__` function after any expensive operations.

Creating an *Effect* allows the effect to be re-used by multiple components.

```
from anvil_extras.animation import Effect, Transition

fade_in = Transition(opacity=[0, 1])
effect = Effect(fade_in, duration=500)

def card_show(self, **event_args):
    effect.animate(self.card)
```

Alternatively use *animate* with a *Transition* and timing options.

```
from anvil_extras.animation import animate, fade_in

def card_show(self, **event_args):
    animate(self.card, fade_in, duration=500)
```

Animate on remove

When a component is removed we need to wait for an animation to complete before removing it.

```
from anvil_extras.animation import animate, fade_out, Easing, Effect

leave_effect = Effect(fade_out, duration=500, easing=Easing.ease_out)


def button_click(self, **event_args):
    if self.card.parent is not None:
        # we can't do this in the hide event because we're already off the screen!
        leave_effect.animate(self.card).wait()
        self.card.remove_from_parent()
```

Combine Transitions

Transitions can be combined with the | operator. They will be merged like dictionaries.

```
from anvil_extras.animation import animate, zoom_out, fade_out, Transition

zoom_fade_out = zoom_out | fade_out
zoom_fade_in = reversed(zoom_fade_out)

def button_click(self, **event_args):
    if self.card.parent is not None:
        t = zoom_fade_out | Transition.height_out(component)
```

(continues on next page)

(continued from previous page)

```
        animate(self.card, t, duration=500).wait()
        self.card.remove_from_parent()
```

Animate on visible change

Some work is needed to animate a Component when the visibility property changes. A helper function might look something like.

```
from anvil_extras.animation import Transition, wait_for

zoom = Transition(scale=[.3, 1], opacity=[0, 1])

def visible_change(self, component):
    if is_animating(component):
        return

    is_visible = component.visible
    if not is_visible:
        # set this now because we need it on the screen to measure its height
        # if you have a show event for this component - it may also fire
        component.visible = True
        direction = "normal"
    else:
        direction = "reverse"

    t = zoom | Transition.height_in(component)
    animate(component, t, duration=900, direction=direction)

    if is_visible:
        # we're animating - wait for the animation to finish before setting visible to␣
↪False
        wait_for(component) # equivalent to animation.wait() or wait_for(animation)
        component.visible = False
```

Swap Elements

Swapping elements requires us to animate from one component to another. We wait for the animation to finish. Then, remove the components and add them back in their new positions. Removing and adding components happens quickly so that the user only sees the components switching places.

```
from anvil_extras.animation import animate

def button_click(self, **event_args):
    # animate wait then remove and re-add
    components = self.linear_panel.get_components()
    c0, c1 = components[0], components[1]
    animate(c0, end_at=c1)
    animate(c1, end_at=c0).wait()
    c0.remove_from_parent()
    c1.remove_from_parent()
```

(continues on next page)

(continued from previous page)

```
    self.linear_panel.add_component(c0, index=0)
    self.linear_panel.add_component(c1, index=0)
```

An alternative version would get the positions of the components. Then remove and add the components to their new positions. Finally animating the components starting from whence they came to their new positions.

```
from anvil_extras.animation import animate, get_bounding_rect, is_animating

def button_click(self, **event_args):
    # get positions, remove, change positions, reverse animate
    components = self.linear_panel.get_components()
    c0, c1 = components[0], components[1]
    if is_animating(c0) or is_animating(c1):
        return
    p0, p1 = get_bounding_rect(c0), get_bounding_rect(c1)
    c0.remove_from_parent()
    c1.remove_from_parent()
    self.linear_panel.add_component(c0, index=0)
    self.linear_panel.add_component(c1, index=0)
    animate(c0, start_at=p0)
    animate(c1, start_at=p1)
```

Switch positions might be useful in a RepatingPanel. Here's what that code might look like.

```
from anvil_extras.animation import animate

class Form1(Form1Template):
    def __init__(self, **properties):
        ...
        self.repeating_panel_1.set_event_handler('x-swap', self.swap)


    def swap(self, component, is_up, **event_args):
        """this event is raised by a child component"""
        items = self.repeating_panel_1.items
        components = self.repeating_panel_1.get_components()
        i = components.index(component)
        j = i - 1 if is_up else i + 1
        if j < 0:
            # we can't go negative
            return
        c1 = component
        try:
            c2 = components[j]
        except IndexError:
            return

        animate(c1, end_at=c2)
        animate(c2, end_at=c1).wait()
        items[i], items[j] = items[j], items[i]
        self.repeating_panel_1.items = items
```

(continues on next page)

(continued from previous page)

```
class ItemTemplate1(ItemTemplate1Template):
    def __init__(self, **properties):
        # Set Form properties and Data Bindings.
        self.init_components(**properties)
        # Any code you write here will run when the form opens.

    def up_btn_click(self, **event_args):
        """This method is called when the button is clicked"""
        self.parent.raise_event('x-swap', component=self, is_up=True)

    def down_btn_click(self, **event_args):
        """This method is called when the button is clicked"""
        self.parent.raise_event('x-swap', component=self, is_up=False)
```

4.1.3 Full API

is_animating(*component*, *include_children=False*)
Returns a boolean as to whether the component is animating. If `include_children` is set to `True` all child elements will also be checked.

wait_for(*component_or_animation*, *include_children=False*)
If given an animation equivalent to `animation.wait()`. If given a component, will wait for all running animations on the component to finish. If `include_children` is set to `True` all child elements will be waited for.

animate(*component*, *transition=None*, *start_at=None*, *end_at=None*, *use_ghost=False*, ***effect_timing_options*)
`component`: an anvil Component or Javascript HTMLElement

`transition`: Transition object

`effect_timing_options`: various options to change the behaviour of the animation e.g. `duration=500`.

`use_ghost`: when set to `True`, will animate a ghost element (i.e. a visual copy). Using a ghost element will allow the component to be animated outside of its container

`start_at`, `end_at`: Can be set to a `Component` or `DOMRect` (i.e. a computed position of a component from `get_bounding_rect`) If either `start_at` or `end_at` are set this will determine the start/end position of the animation If one value is set and the other omitted the omitted value will be assumed to be the current position of the component. A ghost element is always used when `start_at` / `end_at` are set.

get_bounding_rect(*component*)
Returns a `DOMRect` object. A convenient way to get the `height`, `width`, `x`, `y` values of a *component*. Where the `x`, `y` are the absolute positions on the page from the top left corner.

class Transition(*cssProp0=list[str]*, *cssProp1=list[str]*, *transformProp0=list[str]*, *offset=list[int | float]*)
Takes CSS/transform property names as keyword arguments and each value should be a list of frames for that property. The number of frames must match across all transform based properties.

`fly_right = Transition(translateX=[0, "100%"], scale=[1, 0], opacity=[0, 0.25, 1])` is valid since opacity is not a transform property.

`slide_right = Trnasition(translateX=[0, "100%"], scale=[1, 0.75, 0])` is invalid since the `scale` and `translateX` are transform properties with mismatched frame lengths.

Each frame in the list of frames represents a CSS value to be applied across the transition. Typically the first value is the start of the transition and the last value is the end. Lists can be more than 2 values, in which case the transition will be split across the values evenly. You can customize the even split by setting an offset that has values from 0 to 1

```
fade_in_slow = Transition(opacity=[0, 0.25, 1], offset=[0, 0.75, 1])
```

Transition objects can be combined with the | operator (which behaves like merging dictionaries) `t = reversed(slide_right) | zoom_in | fade_in | Transtion.height_in(component)` If two transitions have mismatched frame lengths for transform properties this will fail.

classmethod height_out(*cls*, *component*)
Returns a Transition starting from the current height of the component and ending at 0 height.

classmethod height_in(*cls*, *component*)
Returns a Transition starting from height 0 and ending at the current height of the component.

classmethod width_out(*cls*, *component*)
Returns a Transition starting from the current width of the component and ending at 0 width.

classmethod width_in(*cls*, *component*)
Returns a Transition starting from width 0 and ending at the current width of the component.

reversed(transition)
Returns a Transition with all frames reversed for each property.

Effect(transition, **effect_timing_options):
Create an effect that can later be used to animate a component. The first argument should be a Transition object. Other keyword arguments should be effect timing options.

animate(*self*, *component*, *use_ghost=False*)
animate a component using an effect object. If `use_ghost` is `True` a ghost element will be animated. Returns an Animation instance.

getKeyframes(*self*, *component*)
Returns the computed keyframes that make up this effect. Can be used in place of the `transition` argument in other functions.

getTiming(*self*, *component*)
Returns the EffectTiming object associated with this effect.

Animation(component, effect):
An Animation object will be returned from the `Effect.animate()` method and the `animate()` function. Provides playback control for an animation.

cancel(*self*)
abort animation playback

commitStyles(*self*)
Commits the end styling state of an animation to the element

finish(*self*)
Seeks the end of an animation

pause(*self*)
Suspends playing of an animation

play(*self*)
Starts or resumes playing of an animation, or begins the animation again if it previously finished.

persist(*self*)
Explicitly persists an animation, when it would otherwise be removed.

reverse(*self*)
Reverses playback direction and plays

updatePlaybackRate(*self*, *playback_rate*)
The new speed to set. A positive number (to speed up or slow down the animation), a negative number (to reverse), or zero (to pause).

wait(*self*)
Animations are not blocking. Call the wait function to wait for an animation to finish in a blocking way

playbackRate
gets or sets the playback rate

onfinish
set a callback for when the animation finishes

oncancel
set a callback for when the animation is cancelled

onremove
set a callback for when the animation is removed

Easing
An Enum like instance with some common easing values.

`Easing.ease`, `Easing.ease_in`, `Easing.ease_out`, `Easing.ease_in_out` and `Easing.linear`.

cubic_bezier(*po*, *p1*, *p2*, *p3*)
Create a `cubic_bezier` easing value from 4 numerical values.

4.1.4 Pre-computed Transitions

Attention Seekers

- `pulse = Transition(scale=[1, 1.05, 1])`
- `bounce = Transition(translateY=[0, 0, "-30px", "-30px", 0, "-15px", 0, "-15px", 0], offset=[0, 0.2, 0.4, 0.43, 0.53, 0.7, 0.8, 0.9, 1])`
- `shake = Transition(translateX=[0] + ["10px", "-10px"] * 4 + [0])`

Fades

- `fade_in = Transition(opacity=[0, 1])`
- `fade_in_slow = Transition(opacity=[0, 0.25, 1], offset=[0, 0.75, 1])`
- `fade_out = reversed(fade_in)`

Slides

- slide_in_up = Transition(translateY=["100%", 0])
- slide_in_down = Transition(translateY=["-100%", 0])
- slide_in_left = Transition(translateX=["-100%", 0])
- slide_in_right = Transition(translateX=["100%", 0])
- slide_out_up = reversed(slide_in_down)
- slide_out_down = reversed(slide_in_up)
- slide_out_left = reversed(slide_in_left)
- slide_out_right = reversed(slide_in_right)

Rotate

- rotate = Transition(rotate=[0, "360deg"])

Zoom

- zoom_in = Transition(scale=[.3, 1])
- zoom_out = reversed(zoom_in)

Fly

- fly_in_up = slide_in_up | zoom_in | fade_in
- fly_in_down = slide_in_down | zoom_in | fade_in
- fly_in_left = slide_in_left | zoom_in | fade_in
- fly_in_right = slide_in_right | zoom_in | fade_in
- fly_out_up = reversed(fly_in_down)
- fly_out_down = reversed(fly_in_up)
- fly_out_left = reversed(fly_in_left)
- fly_out_right = reversed(fly_in_right)

4.2 Augmentation

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

4.2.1 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'
```

4.2.2 need a trigger method?

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

4.2.3 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)
```

4.2.4 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
```

4.2.5 DataGrid pagination_click

Importing the augment module gives DataGrid's a `pagination_click` event

```
self.data_grid.set_event_handler('pagination_click', self.pagination_click)

def pagination_click(self, **event_args):
    button = event_args["button"] # 'first', 'last', 'previous', 'next'
    print(button, "was clicked")
```

4.3 Authorisation

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

4.3.1 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

4.3.2 Usage

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

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
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 order of the decorators matters. *anvil.server.callable* must come before either of the authorisation module decorators.

4.4 Messaging

4.4.1 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.

4.4.2 Usage

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 anvil_extras.messaging import Publisher

publisher = Publisher()
```

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

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.

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):
```

(continues on next page)

(continued from previous page)

```
        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.

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 anvil_extras.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.

4.5 Navigation

A client module for that provides dynamic menu construction.

4.5.1 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!

4.5.2 Usage

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)
```

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 = [
  {"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 anvil_extras 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",
```

(continues on next page)

(continued from previous page)

```
    "icon": "fa:gear",
    "visibility": {
      "x-advanced-mode-enabled": True,
      "x-advanced-mode-disabled": False
    }
  }
]


@routing.main_router
class Main(MainTemplate):

  def __init__(self, **properties):
    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 on the Main class.

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())
```

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):
```

(continues on next page)

(continued from previous page)

```
  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)
```

Navigate with Code

You can emulate clicking a menu link using the go_to function, which takes a 'target' key as its only parameter, e.g.

```
navigation.go_to("contact")
```

4.6 Popovers

A client module that allows bootstrap popovers in anvil

Live Example: popover-example.anvil.app

Example Clone Link:

4.6.1 Introduction

Popovers are already included with anvil since anvil ships with bootstrap.

This module provides a python wrapper around bootstrap popovers. When the `popover` module is imported, all anvil components get two additional methods - `pop` and `popover`.

4.6.2 Usage

```
from anvil_extras import popover
# importing the module adds the popover method to Button and Link

    self.button = Button()
    self.button.popover(content='example text', title='Popover', placement="top")
```

```
from anvil_extras import popover

    self.button_1.popover(Form2(), trigger="manual")
```

(continues on next page)

(continued from previous page)

```
    # content can be an anvil component

    def button_1_click(self, **event_args):
        if self.button_1.pop("is_visible"):
            self.button_1.pop("hide")
        else:
            self.button_1.pop("show")
        # equivalent to self.button_1.pop("toggle")
```

4.6.3 API

popover(*self*, *content*, *title=''*, *placement='right'*, *trigger='click'*, *animation=True*, *delay={'show': 100, 'hide': 100}*, *max_width=None*, *auto_dismiss=True*)

popover is a method that can be used with any anvil component. Commonly used on `Button` and `Link` components.

self
: the component used. No need to worry about this argument when using popover as a method e.g. `self.button_1.popover(content='example text')`

content
: content can be a string or an anvil component. If an anvil component is used - that component will have a new attribute `popper` added. This allows the the content form to close itself using `self.popper.pop('hide')`.

title
: optional string.

placement
: One of `'right'`, `'left'`, `'top'`, `'bottom'`. If using `left` or `right` it may be best to place the component in a `FlowPanel`.

trigger
: One of `'manual'`, `'focus'`, `'hover'`, `'click'`, (can be a combination of two e.g. `'hover focus'`). `'stickyhover'` is also available.

animation
: `True` or `False`

delay
: A dictionary with the keys `'show'` and `'hide'`. The values for `'show'` and `'hide'` are in milliseconds.

max_width
: bootstrap default is 276px you might want this wider

auto_dismiss
: When clicking outside a popover the popover will be closed. Setting this flag to `False` overrides that behaviour. Note that popovers will always be dismissed when the page is scrolled. This prevents popovers from appearing in weird places on the page.

pop(*self*, *behaviour*)

pop is a method that can be used with any component that has a `popover`

self
: the component used. No need to worry about this argument when using `self.button_1.pop('show')`

behaviour
'show', 'hide', 'toggle', 'destroy'. Also includes 'shown' and 'is_visible', which return a boolean. 'update' will update the popover's position. This is useful when a popover's height changes dynamically.

dismiss_on_outside_click(*dismiss=True*)
by default if you click outside of a popover the popover will close. This behaviour can be overridden globally by calling this function. It can also be set per popover using the auto_dismiss argument. Note that popovers will always be dismissed when the page is scrolled. This prevents popovers from appearing in weird places on the page.

set_default_max_width(*width*)
update the default max width - this is 276px by default - useful for wider components.

has_popover(*component*)
Returns a bool as to whether the component has a popover. A useful flag to prevent creating unnecessary popovers.

4.7 Serialisation

A server module that provides dynamic serialisation of data table rows.

A single data table row is converted to a dictionary of simple Python types. A set of rows is converted to a list of those dictionaries.

4.7.1 Usage

Let's imagine we have a data table named 'books' with columns 'title' and 'publication_date'.

In a server module, import and call the function *datatable_schema* to get a marshmallow Schema instance:

```
from anvil.tables import app_tables
from anvil_extras.serialisation import datatable_schema
from pprint import pprint

schema = datatable_schema("books")
```

To serialise a row from the books table, call the schema's *dump* method:

```
book = app_tables.books.get(title="Fluent Python")
result = schema.dump(book)
pprint(result)

>> {"publication_date": "2015-08-01", "title": "Fluent Python"}
```

To serialise several rows from the books table, set the *many* argument to True:

```
books = app_tables.books.search()
result = schema.dump(books, many=True)
pprint(result)

>> [{'publication_date': '2015-08-01', 'title': 'Fluent Python'},
>>  {'publication_date': '2015-01-01', 'title': 'Practical Vim'},
>>  {'publication_date': None, 'title': "The Hitch Hiker's Guide to the Galaxy"}]
```

To exclude the publication date from the result, pass its name to the server function:

```
from anvil.tables import app_tables
from anvil_extras.serialisation import datatable_schema
from pprint import pprint

schema = datatable_schema("books", ignore_columns="publication_date")
books = app_tables.books.search()
result = schema.dump(books, many=True)
pprint(result)

>> [{'title': 'Fluent Python'},
>>  {'title': 'Practical Vim'},
>>  {'title': "The Hitch Hiker's Guide to the Galaxy"}]
```

You can also pass a list of column names to ignore.

If you want the row id included in the results, set the *with_id* argument:

```
from anvil.tables import app_tables
from anvil_extras.serialisation import datatable_schema
from pprint import pprint

schema = datatable_schema("books", ignore_columns="publication_date", with_id=True)
books = app_tables.books.search()
result = schema.dump(books, many=True)
pprint(result)

>> [{'_id': '[169162,297786594]', 'title': 'Fluent Python'},
>>  {'_id': '[169162,297786596]', 'title': 'Practical Vim'},
>>  {'_id': '[169162,297786597]',
>>   'title': "The Hitch Hiker's Guide to the Galaxy"}]
```

Linked Tables

Let's imagine we also have an 'authors' table with a 'name' column and that we've added an 'author' linked column to the books table.

To include the author in the results for a books search, create a dict to define, for each table, the linked columns in that table the linked table they refer to:

```
from anvil.tables import app_tables
from anvil_extras.serialisation import datatable_schema
from pprint import pprint

# The books table has one linked column named 'author' and that is a link to the 'authors'␣
↪table
linked_tables = {"books": {"author": "authors"}}
schema = datatable_schema(
    "books",
    ignore_columns="publication_date",
    linked_tables=linked_tables,
 )
books = app_tables.books.search()
```

(continues on next page)

(continued from previous page)

```
result = schema.dump(books, many=True)
pprint(result)

>> [{'author': {'name': 'Luciano Ramalho'}, 'title': 'Fluent Python'},
>>  {'author': {'name': 'Drew Neil'}, 'title': 'Practical Vim'},
>>  {'author': {'name': 'Douglas Adams'},
>>   'title': "The Hitch Hiker's Guide to the Galaxy"}]
```

Finally, let's imagine the 'authors' table has a 'date_of_birth' column but we don't want to include that in the results:

```
from anvil.tables import app_tables
from anvil_extras.serialisation import datatable_schema
from pprint import pprint

linked_tables = {"books": {"author": "authors"}}
ignore_columns = {"books": "publication_date", "authors": "date_of_birth"}
schema = datatable_schema(
    "books",
    ignore_columns=ignore_columns,
    linked_tables=linked_tables,
 )
books = app_tables.books.search()
result = schema.dump(books, many=True)
pprint(result)

>> [{'author': {'name': 'Luciano Ramalho'}, 'title': 'Fluent Python'},
>>  {'author': {'name': 'Drew Neil'}, 'title': 'Practical Vim'},
>>  {'author': {'name': 'Douglas Adams'},
>>   'title': "The Hitch Hiker's Guide to the Galaxy"}]
```

4.8 Storage

4.8.1 Introduction

Browsers have various mechanisms to store data. `localStorage` and `IndexedDB` are two such mechanisms. These are particularly useful for storing data offline.

The anvil_extras `storage` module provides wrappers around both these storage mechanisms in a convenient dictionary like API.

In order to store data you'll need a store object. You can import the default store objects `local_storage` or `indexed_db`. Alternatively create your own store object using the classmethod `create_store(store_name)`.

NB: when working in the IDE the app is running in an IFrame and the storage objects may not be available. This can be fixed by changing your browser settings. Turning the shields down in Brave or making sure not to block third party cookies in Chrome should fix this.

Which to chose?

If you have small amounts of data which can be converted to JSON - use `local_storage`.
If you have more data which can be converted to JSON (also `bytes`) - use `indexed_db`.

`datetime` and `date` objects are also supported. If you want to store anything else you'll need to convert it to something JSONable first.

4.8.2 Usage Examples

Store user preference

```
from anvil_extras.storage import local_storage

class UserPreferences(UserPreferencesTemplate):
    def __init__(self, **properties):
        self.init_components(**properties)

    def dark_mode_checkbox_change(self, **event_args):
        local_storage['dark_mode'] = self.dark_mode_checkbox.checked
```

Change the theme at startup

```
## inside a startup module
from anvil_extras.storage import local_storage

if local_storage.get('dark_mode') is not None:
    # set the app theme to dark
    ...
```

Create an offline todo app

```
from anvil_extras.storage import indexed_db
from anvil_extras.uuid import uuid4

todo_store = indexed_db.create_store('todos')
# create_store() is a classmethod that takes a store_name
# it will create another store object inside the browsers IndexedDB
# or return the store object if it already exists
# the todo_store acts as dictionary like object

class TodoPage(TodoPageTemplate):
    def __init__(self, **properties):
        self.init_components(**properties)
        self.todo_panel.items = list(todo_store.values())

    def save_todo_btn_click(self, **event_args):
        if not self.todo_input.text:
```

(continues on next page)

(continued from previous page)

```
            return
        id = str(uuid4())
        todo = {"id": id, "todo": self.todo_input.text, "completed": False}
        todo_store[id] = todo
        self.todo_panel.items = self.todo_panel.items + [todo]
        self.todo_input.text = ""
```

4.8.3 API

class StorageWrapper
class IndexedDBWrapper
class LocalStorageWrapper

both `indexed_db` and `local_storage` are instances of the dictionary like classes *IndexedDBWrapper* and *LocalStorageWrapper* respectively.

classmethod create_store(*name*)

Create a store object. e.g. `todo_store = indexed_db.create_store('todos')`. This will create a new store inside the browser's `IndexedDB` and return an *IndexedDBWrapper* instance. The `indexed_db` object is equivalent to `indexed_db.create_store('default')`. To explore this further, open up devtools and find `IndexedDB` in the Application tab. Since *create_store* is a classmethod you can also do `todo_store = IndexedDBWrapper.create_store('todos')`.

is_available()

Check if the storage object is supported. Returns a `boolean`.

list(store)

Return a list of all the keys used in the *store*.

len(store)

Return the number of items in *store*.

store[key]

Return the value of *store* with key *key*. Raises a `KeyError` if *key* is not in *store*.

store[key] = value

Set `store[key]` to *value*. If the value is not a JSONable data type it may be stored incorrectly. If storing `bytes` objects it is best to use the `indexed_db` store. `datetime` and `date` objects are also supported.

del store[key]

Remove `store[key]` from *store*.

key in store

Return `True` if *store* has a key *key*, else `False`.

iter(store)

Return an iterator over the keys of the *store*. This is a shortcut for `iter(store.keys())`.

clear()

Remove all items from the *store*.

get(*key*[, *default*])

Return the value for *key* if *key* is in *store*, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

items()

Return an iterator of the *store*'s `(key, value)` pairs.

keys()
Return an iterator of the *store*'s keys.

pop(*key*[, *default*])
If *key* is in *store*, remove it and return its value, else return *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

store(*key*, *value*)
Equivalent to `store[key] = value`.

update([*other*])
Update the *store* with the key/value pairs from *other*, overwriting existing keys. Return `None`.

update() accepts either a dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, *store* is then updated with those key/value pairs: `store.update(red=1, blue=2)`.

values()
Return an iterator of the *store*'s values.

4.9 Utils

Client and server side utility functions.

4.9.1 Timing

There are client and server side decorators which you can use to show that a function has been called and the length of time it took to execute.

Client Code

Import the `timed` decorator and apply it to a function:

```
from anvil_extras.utils import timed

@timed
def target_function(args, **kwargs):
    print("hello world")
```

When the decorated function is called, you will see messages in the IDE console showing the arguments that were passed to it and the execution time.

Server Code

Import the `timed` decorator and apply it to a function:

```
import anvil.server
from anvil_extras.server_utils import timed


@anvil.server.callable
@timed
```

(continues on next page)

(continued from previous page)

```
def target_function(args, **kwargs):
    print("hello world")
```

On the server side, the decorator takes a `logging.Logger` instance as one of its optional arguments. The default instance will log to stdout, so that messages will appear in your app's logs and in the IDE console. You can, however, create your own logger and pass that instead if you need more sophisticated behaviour:

```
import logging
import anvil.server
from anvil_extras.server_utils import timed

my_logger = logging.getLogger(__name__)


@timed(logger=my_logger)
def target_function(args, **kwargs):
    ...
```

The decorator also takes an optional `level` argument which must be one of the standard levels from the logging module. When no argument is passed, the default level is `logging.INFO`.

4.9.2 Auto-Refresh

Whenever you set a form's `item` attribute, the form's `refresh_data_bindings` method is called automatically.

The `utils` module includes a decorator you can add to a form's class so that `refresh_data_bindings` is called whenever `item` changes at all.

To use it, import the decorator and apply it to the class for a form:

```
from anvil_extras.utils import auto_refreshing
from ._anvil_designer import MyFormTemplate


@auto_refreshing
class MyForm(MyFormTemplate):
    def __init__(self, **properties):
        self.init_components(**properties)
```

Now, the form has an `item` property which behaves like a dictionary. Whenever a value of that dictionary changes, the form's `refresh_data_bindings` method will be called.

Note: The `item` property will no longer reference the same object. Rather, in the following example, it is as though auto-refresh adds the `item = dict(item)` line:

```
other_item = {"x": 1}
item = other_item

item = dict(item)
item["x"] = 2
```

As in the above code, with auto-refresh, `item` is changed but `other_item` is not.

4.9.3 Wait for writeback

Using `wait_for_writeback` as a decorator prevents a function executing before any queued writebacks have completed.

This is particularly useful if you have a form with text fields. Race condidtions can occur between a text field writing back to an item and a click event that uses the item.

To use `wait_for_writeback`, import the decorator and apply it to a function, usually an event_handler:

```
from anvil_extras.utils import wait_for_writeback

class MyForm(MyFormTemplate):
    ...

    @wait_for_writeback
    def button_1_click(self, **event_args):
        anvil.server.call("save_item", self.item)
```

The click event will now only be called after all active writebacks have finished executing.

4.9.4 Correct Canvas Resolution

Canvas elements can appear blurry on retina screens. This helper function ensures a canvas element appears sharp. It should be called inside the canvas `reset` event.

```
from anvil_extras.utils import correct_canvas_resolution

class MyForm(MyFormTemplate):
    ...

    def canvas_reset(self, **event_args):
        c = self.canvas
        correct_canvas_resolution(c)
        ...
```

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

T

U

V

W