# Anvil Extras

The Anvil Extras project team

May 31, 2023

# CONTENTS


**1 Installation** **1**
- 1.1 Install as a third-party dependency . . . 1
- 1.2 Install as a clone . . . 1

**2 Contributing** **3**
- 2.1 Issues . . . 3
- 2.2 Commits . . . 3
- 2.3 Components . . . 3
- 2.4 Python Code . . . 4
- 2.5 Documentation . . . 4
- 2.6 Testing . . . 4
- 2.7 Merging . . . 4
- 2.8 Copyright . . . 4

**3 Components** **5**
- 3.1 Autocomplete . . . 5
- 3.2 Chip . . . 5
- 3.3 ChipsInput . . . 6
- 3.4 Determinate ProgressBar . . . 7
- 3.5 EditableCard . . . 8
- 3.6 Indeterminate ProgressBar . . . 8
- 3.7 MessagePill . . . 8
- 3.8 MultiSelectDropdown . . . 9
- 3.9 PageBreak . . . 11
- 3.10 Pivot . . . 12
- 3.11 Quill Editor . . . 12
- 3.12 Slider . . . 14
- 3.13 Switch . . . 18
- 3.14 Tabs . . . 18

**4 Modules** **21**
- 4.1 Animation . . . 21
- 4.2 Augmentation . . . 28
- 4.3 Authorisation . . . 30
- 4.4 Hashlib . . . 31
- 4.5 Logging . . . 32
- 4.6 Messaging . . . 35
- 4.7 Navigation . . . 37
- 4.8 NonBlocking . . . 40
- 4.9 Persistence . . . 43

4.10 Popovers . . . 47
4.11 Routing . . . 50
4.12 Serialisation . . . 69
4.13 Storage . . . 72
4.14 Utils . . . 75
4.15 Zod . . . 77

**5 Indices and tables** **93**

**Index** **95**

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

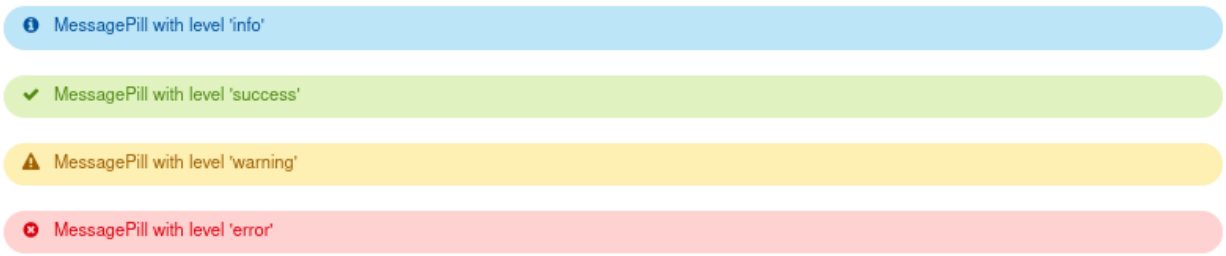
The colour of the background track

**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 Overrides

**format_selected_text**(*self*, *count*, *total*)

This method is called when the selection changes and should return a string.

The default implementation looks like:

```
from anvil_extras import MultiSelectDropdown

def format_selected_text(self, count, total):
    if count > 3:
        return f"{count} items selected"
    return ", ".join(self.selected_keys)
```

You can change this by overriding this method.

You can override it globally by doing the following

```
from anvil_extras import MultiSelectDropdown

def format_selected_text(self, count, total):
    if count > 2:
        return f"{count} items selected of {total}"
    return ", ".join(self.selected_keys)

MultiSelectDropdown.format_selected_text = format_selected_text
```

Alternatively you can change the `count_selected_text` method per multiselect instance

```
class Form1(Form1Template):
    def __init__(self, **properties):
        ...

        def format_selected_text(count, total):
            if count > 3:
                return f"{count} items selected"
            return ", ".join(self.multi_select_drop_down_1.selected_keys)
```

(continues on next page)

(continued from previous page)

```
        self.multi_select_drop_down_1.format_selected_text = format_selected_text
```

### 3.8.2 Properties

**align**
String

"left", "right", "center"

**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

**width**
String | Number

The default width is 200px. This can be set using any css length. Alternatively set the width to be `"auto"`, which will adjust the width to be as wide as the largest option. `"fit"` (or `"fit-content"`) will size the dropdown depending on what is selected. Use width `"100%""` to make the dropdown fill its container.

**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.

**enable_select_all**
Boolean

Enable Select All and Deselect All buttons.

### 3.8.3 Events

**change**
When the selection changes

**opened**
When the dropdown is opened

**closed**
When the dropdown is closed

**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.

**sanitize**
Boolean

Set the default sanitize behaviour used for the `set_html()` method.

**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(html, sanitize=None)**
: Set the contents of the Quill editor to html. If `sanitize` is `True`, then the html will be sanitized in the same way that a RichText component sanitizes the html. If `sanitize` is unset the the default `sanitize` attribute will be used to determine this behaviour. If 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'`.

**color**
str

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

**bar_height**
str | int

The height of the bar. Can be a css length or an integer, which will be set to the pixel height. Defaults to 18.

**handle_size**
str

The size of the handle. Can be a css length or an integer, which will be the diameter of the handle. Defaults to 34.

**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. Includes the parameters `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 Hashlib

A client module that provides several hashing algorithms.

### 4.4.1 Usage

The module provides the functions `sha1`, `sha256`, `sha384` and `sha512`. Each can be called by passing the str or bytes object to be hashed and will return a hex string.

e.g.

```
from anvil_extras.hashlib import sha256

print(sha256("Hello World!"))

>>> 7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069
```

## 4.5 Logging

A lightweight logging implementation, similar to Python's logging module. It can be used on both the server and the client. It supports logging levels and a custom format.

### 4.5.1 Logger

```
from anvil_extras.logging import Logger, DEBUG

user_logging = Logger(
    name="user",
    level=DEBUG,
    format="{name}-{level} {datetime:%Y-%m-%d %H:%M:%S}: {msg}",
)

user_logging.info("user logging ready")
# outputs 'user-INFO 2022-01-01 12:00:00: user logging ready'
```

#### API

**class Logger**(*name='root'*, *level=logging.INFO*, *format='{name}: {msg}'*, *stream=sys.stdout*)

**name**

The name of the logger. Useful for distinguishing loggers in app logs.

**level**

One of `logging.NOTSET`, `logging.DEBUG`, `logging.INFO`, `logging.WARNING`, `logging.CRITICAL` If the logging level is set to `logging.INFO`, then only logs at the level of `INFO`, `WARNING` or `CRITICAL` will be logged. This is useful for turning on and off debug logs in your app.

**format**

A format string. Valid options include `name`, `level`, `msg`, `datetime`, `time`, `date`.

**stream**

A valid stream is any object that has a valid `.write()` and `.flust()` method. The default stream is the `sys.stdout` stream. This will log to the console in the IDE and get passed to the app logs. A valid python stream can be used. On the client, you may want to create your own.

**disabled**

To stop a logger from outputting to the console set it to disabled `logger.disabled = True`.

```
class CustomStream:
    def __init__(self, lbl):
        self.lbl = lbl

    def write(self, text):
        self.lbl.text += text

    def flush(self):
        pass
```

**log**(*level*, *msg*)

The level is a valid logging level. If the level is greater than or equal to the logger's level the msg will be logged according to the logger's format.

**debug**(*msg*)

Equivalent to `logger.log(logging.DEBUG, msg)`

**info**(*msg*)

Equivalent to `logger.log(logging.INFO, msg)`

**warning**(*msg*)

Equivalent to `logger.log(logging.WARNING, msg)`

**error**(*msg*)

Equivalent to `logger.log(logging.ERROR, msg)`

**critical**(*msg*)

Equivalent to `logger.log(logging.CRITICAL, msg)`

**get_format_params**(*, *level*, *msg*, ***params*)

This method can be overridden by a subclass. Any extra params can be used in the format string.

```
class TimerLogger(Logger):
    def get_format_params(self, **params):
        elapsed = time.time() - self.curr_time
        return super().get_format_params(elapsed=elapsed, **params)

# with UID

from anvil_extras.uuid import uuid4

class UIDLogger(Logger):
    def __init__(self, name="uid logger", uid=None, level=INFO, format="{uid}: {msg}
↪"):
        super().__init__(name=name, level=level, format=format)
        self.uid = uid or uuid4()

    def get_format_params(self, **params):
        return super().get_format_params(uid=self.uid, **params)
```

### 4.5.2 TimerLogger

The `TimerLogger` is a subclass of `Logger` and allows for debug timing in various ways. It supports an extra format argument `elapsed`. The default format for a `TimerLogger` is: `"{time:%H:%M:%S} | {name}: ({elapsed:6.3f} secs) | {msg}"`

It adds 3 methods to the API above:

**start**(*msg='start'*)

records the starting timestamp

**check**(*msg='check'*, *restart=False*)

records the elapsed time (optionally restart the `TimerLogger`)

**end**(*msg='end'*)

records the elapsed time and ends the `TimerLogger`

The `TimerLogger` can be used to check times between lines of code.

```
from anvil_extras.logging import TimerLogger
from time import sleep

T = TimerLogger("my timer")
T.start("starting") # optional msg
sleep(1)
T.check("first check") # optional msg
sleep(3)
T.check("second check", restart=True) # restarts the timer
sleep(2)
T.end() # optional msg - ends the timer
```

The above code logs:

```
# 20:57:56 | my timer: ( 0.000 secs) | starting
# 20:57:57 | my timer: ( 1.012 secs) | first check
# 20:58:00 | my timer: ( 4.020 secs) | second check (restart)
# 20:58:02 | my timer: ( 2.005 secs) | end
```

Each method can take an optional msg argument. Each method calls the the `.debug()` method, i.e. if you set `TimerLogger(level=logging.INFO)`, then the above logs would not be displayed in the console.

A `TimerLogger` can be used with a `with` statement (as a context manager).

```
from anvil_extras.logging import TimerLogger
from time import sleep

def foo():
    with TimerLogger("timing foo") as T:
        sleep(1)
        T.check("first check")
        sleep(3)
        T.check("second check", restart=True)
        sleep(2)
```

When used as a context manager the `TimerLogger` will call the `.start()` and `.end()` method.

The `TimerLogger` can be used as a convenient decorator.

```
from anvil_extras.logging import TimerLogger
from time import sleep

@TimerLogger("foo timer")
def foo():
    ...

foo()

# 21:12:47 | foo timer: ( 0.000 secs) | start
# 21:12:48 | foo timer: ( 1.014 secs) | end
```

For a more detailed timing decorator use `anvil_extras.utils.timed` decorator.

## 4.6 Messaging

### 4.6.1 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.

### 4.6.2 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):
```

(continues on next page)

(continued from previous page)

```
        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):
        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.

## 4.7 Navigation

A client module for that provides dynamic menu construction.

### 4.7.1 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!

### 4.7.2 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",
    "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):
  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")
```

## 4.8 NonBlocking

Call functions in a non-blocking way.

In a blocking execution, the next line of code will not be executed until the current line has completed.

In contrast, non-blocking execution allows the next line to be executed without waiting for the current line to complete.

---

**Note:** This module cannot be used to call server functions simultaneously, as Anvil server calls are queued.

---

A suitable use case for this library is when you want to perform an action without waiting for a response, such as updating a database after making changes on the client side.

### 4.8.1 Examples

#### Call a server function

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

```
from anvil_extras.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)
```

#### Handle return values and errors

If you want to handle the return value or any errors, you can provide result and error handlers.

```
from anvil_extras.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)
```

(continues on next page)

(continued from previous page)

```
    async_call.on_result(self.handle_result)
    async_call.on_error(self.handle_error)
```

### repeat

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

```
from anvil_extras 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_extras import non_blocking

class Form1(Form1Template):
    def __init__(self, **properties):
        ...
        self.deferred_search = None

    def update_search_results(self):
        search_results = anvil.server.call_s("search_results", self.search_box.text)
        # do something with search_results

    def search_box_change(self, **event_args):
        # cancel the existing deferred_search
        non_blocking.cancel(self.deferred_search)
        self.deferred_search = non_blocking.defer(self.update_search_results, 0.3)
```

In this example we call `self.update_search_results()` only when the user has stopped typing for 0.3 seconds. If the user starts typing again before 0.3 seconds is up, the deferred call is cancelled. This prevents us calling the server too often.

### 4.8.2 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 the 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 instantiate this class directly; instead, use the functions above.

**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. Raises an exception if the non-blocking call raised an exception.

**property result**

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

If the non-blocking call has completed, returns the result. Raises an exception if the non-blocking call raised an exception.

**property error**

If the non-blocking call has not yet completed, raises 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 status**

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

**cancel**(*ref*)

Cancel an active call to `delay` or `defer`. The first argument should be `None` or 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 arguments.
- `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 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 arguments.
- `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.

## 4.9 Persistence

Define simple classes for use in client side code and have instances of those classes synchronised with data tables rows.

### 4.9.1 Example

Let's say we have an app that displays books. It has two tables, author and book, with columns:

```
author
    name: text

book
    title: text
    author: linked_column (to author table)
```

The author table contains a row whose name is "Luciano Ramalho" and the book table a row with the title "Fluent Python" and author linked to the row in the author table.

Using the persistence module, we can now define a class for book objects:

```
from anvil_extras.persistence import persisted_class


@persisted_class
class Book:
     key = "title"
```

The data table must have a column with unique entries for each row and we define which that is using the *key* attribute. In this case, we'll assume every book has a unique title.

We can now use that class by creating an instance and telling it to fetch the associated row from the database:

```
book = Book.get("Fluent Python")
```

our *book* object will automatically have each of the row's columns as an attribute:

```
assert book.title == "Fluent Python"
```

But what if we wanted our *book* object to include some information from the author table?

There are two ways to go about that: using a LinkedAttribute or a LinkedClass.

### LinkedAttribute

We can use a *LinkedAttribute* to fetch data from the linked row and include it as an attribute on our object. Let's include the author's name as an attribute of a book:

```
from anvil_extras.persistence import persisted_class, LinkedAttribute


@persisted_class
class Book:
    key = "title"
    author_name = LinkedAttribute(linked_column="author", linked_attr="name")


 book = Book.get("Fluent Python")

 assert book.author_name == "Luciano Ramalho"
```

### LinkedClass

Alternatively, we can define another persisted class for author objects and use an instance of that class as an attribute of a Book:

```
from anvil_extras.persistence import persisted_class

@persisted_class
class Author:
    key = "name"


@persisted_class
class Book:
    author = Author


book = Book.get("Fluent Python")

assert book.author.name == "Luciano Ramalho"
```

#### Customisation

We can, of course, add whatever methods we want to our class. Let's add a property to display the title and author of the book as a single string:

```
from anvil_extras.persistence import persisted_class, LinkedAttribute


@persisted_class
class Book:
    key = "title"
    author_name = LinkedAttribute(linked_column="author", linked_attr="name")

    @property
    def display_text(self):
        return f"{self.title} by {self.author_name}"

book = Book.get("Fluent Python")

assert book.display_text == "Fluent Python by Luciano Ramalho"
```

*NOTE* If you create attributes with leading underscores, they will not form part of any update sent to a server function.

### 4.9.2 Getting and Searching

In the example above, we used the *get* method to fetch a single data table row from the database and create a *Book* instance from it.

For that to work, there needs to be a server function that takes the Book's key as an argument and returns a single row. e.g.:

```
import anvil.server
from anvil.tables import app_tables


@anvil.server.callable
def get_book(title):
    return app_tables.book.get(title=title)
```

The server function's name must be the word *get* followed by the class name in snake case. If we had a class named *MyVeryInterestingThing*, we would need a server function named *get_my_very_interesting_thing*.

Often, we'll want to search for a set of data table rows that meet some criteria and create the resulting instances from the results. For that, we use the *search* method.

Let's assume the book table also has a *publisher* text column. To create a list of books published by O'Reilly we'd call *Book.search* on the client side:

```
books = Book.search(publisher="O'Reilly")
```

and, on the server side, we'd need a function named *search_book* that takes search criteria as arguments and returns a SearchIterator. e.g.:

```
import anvil.server
from anvil.tables import app_tables


@anvil.server.callable
def search_book(*args, **kwargs):
    return app_tables.book.search(*args, **kwargs)
```

The server function name follows the same format as for *get* - it must be the word *search* followed by the class name in snake case.

### 4.9.3 Adding, Updating and Deleting

There are also methods for sending changes to the server - adding new rows, updating and deleting existing rows.

To add a new book, create a Book instance client side and call its *add* method:

```
book = Book(title="JavaScript: The Definitive Guide")
book.add()
```

on the server side, we need a *add_book* function that takes a dict of attribute values as its argument and returns the data table row it creates:

```
import anvil.server
from anvil.tables import app_tables


@anvil.server.callable
def add_book(attrs):
    return app_tables.book.add_row(**attrs)
```

There are similar methods to update or delete an existing row. Let's create a new book, change its title and then delete it:

```
book = Book(title="My Wonderful Book")
book.add()

book.title = "My Not So Wonderful Book"
book.update()

book.delete()
```

As you change an object's attribute values, persistence keeps track of those changes. Calling *update* will send to the server the relevant data table row along with a dict of the changed attribute values. The dict does not contain any attribute whose value has remained unchanged from the underlying row.

So, on the server side, we need *update_book* and *delete_book* functions. The update function must take a data table row and a dict of attribute values as its arguments. The delete function must take a data table row. Neither function needs to return anything:

```
import anvil.server
from anvil.tables import app_tables


```

(continues on next page)

(continued from previous page)

```
@anvil.server.callable
def update_book(row, attrs):
    row.update(**attrs)


@anvil.server.callable
def delete_book(row):
    row.delete()
```

Any additional arguments passed to the *add*, *update* or *delete* methods will be passed to the relevant server function.

### 4.9.4 Caching

Calling the *get* method will attempt to retrieve the matching object from a cache maintained by the persisted class. If there's no cached entry, the relevant server call is made and the resulting object added to the cache.

For the *search* method, the default behaviour is to clear the cache, add entries for each of the objects found and return a list of those results. This behaviour can be disabled by setting the *lazy* argument of the method to *True* whereby the cache is left unaltered and the method will instead return a generator of the objects found.

e.g. in our search example above, we used the default behaviour to return a list of books published by O'Reilly. If, instead, we wanted a generator of those books:

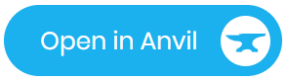
```
books = Book.search(lazy=True, publisher="O'Reilly")
```

## 4.10 Popovers

A client module that allows bootstrap popovers in anvil

Live Example: popover-example.anvil.app

Example Clone Link:




### 4.10.1 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`.

### 4.10.2 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")
    # 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")
```

### 4.10.3 API

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

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 content form to close itself using `self.popper.pop('hide')`.

**title**

optional string.

**placement**

One of `'right'`, `'left'`, `'top'`, `'bottom'` or `'auto'`. If using `left` or `right` it may be best to place the component in a `FlowPanel`. `'auto'` can be combined with other values e.g. `'auto bottom'`.

**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. Note this is ignored if `dismiss_on_outside_click()` is used to set the global behaviour to `False`

**dismiss_on_scroll**

All popovers are hidden when the page is scrolled. See the `dismiss_on_scroll` function for more details. Setting this to `False` may not be what you want unless you've adjusted the container of the popover. This argument will be ignored if set globally to `False` using `dismiss_on_scroll(dismiss=False)`.

**container**

Set the container of the popover to an element or selector on the page. The default value is `"body"`.

**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.

**dismiss_on_scroll**(*dismiss=True*)

By default, if you scroll the popover will close. This behaviour can be overridden globally by calling this function. It can also be set per popover using the `dismiss_on_scroll` argument. Note that popovers will not scroll with their parents by default since they are fixed on the body of the page. If you use this method it should be combined with either, setting the default container to something other than `"body"`.

**set_default_container**(*selector_or_element*)

The default container is `"body"`. This is used since it prevents overflow issues with popovers nested in the anvil component hierarchy. However, it does prevent popovers from scrolling with their attached elements. If you want your popovers to scroll with their popper element, either change this setting globally or use the `container` argument per popover.

**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.

### 4.10.4 Scrolling in Material Design

To support scrolling in Material Design the container element should be a div element within the standard-page.html. It should be nested within the `.content` div.

You can adjust the HTML as follows.

```
<div class="content">
    <div anvil-slot-repeat="default" class="anvil-measure-this"></div>
    <div class="placeholder drop-here" anvil-if-slot-empty="default" anvil-drop-slot=
↪"default">Drop a ColumnPanel here.</div>
    <div id="popover-container" style="position:relative;"></div>
</div>
```

```
from anvil_extras import popover

popover.set_default_container("#popover-container")
popover.dismiss_on_scroll(False)
```

Alternatively you could dynamically insert the container component in your MainForm with python. (Assuming your main form uses the standard-page.html)

```
import anvil.js
from anvil.js.window import document
from anvil_extras import popover


popover_container = document.createElement("div")
popover_container.style.position = "relative"
popover.set_default_container(popover_container)
popover.dismiss_on_scroll(False)


class MainForm(MainFormTemplate):
    def __init__(self, **event_args):
        content_div = anvil.js.get_dom_node(self).querySelector(".content")
        content_div.appendChild(popover_container)
```

## 4.11 Routing

The routing module allows hash-based navigation in an Anvil app.

| | |
|---|---|
| Live Example: | hash-routing-example.anvil.app |
| Example Clone Link: | Example |

### 4.11.1 Introduction

An Anvil app is a single-page app. When the user navigates through the app's pages the URL does not change. The part of the URL before the # is used by the server to identify the app. The part following the #, is never sent to the server and used only by the browser.

The routing module takes advantage of the URL hash and allows unique URLs to be defined for forms within an app. Here are a few examples of URL hashes within an app and associated terminology.

| URL | url_hash | url_pattern | url_dict | url_keys | dynamic_vars |
|---|---|---|---|---|---|
| blog.anvil.app/# | '' | '' | {} | [] | {} |
| blog.anvil.app/#blog | 'blog' | 'blog' | {} | [] | {} |
| blog.anvil.app/#blog?id=10 | 'blog?id=10' | 'blog' | {'id':'10'} | ['id'] | {} |
| blog.anvil.app/#blog/10 | 'blog/10' | 'blog/{id}' | {} | [] | {'id': 10} |

### 4.11.2 Template Forms

These are top-level forms.

A `TemplateForm` is **not** the `HomeForm`. A `TemplateForm` has **no content**. It only has a navigation bar, header, optional sidebar and a `content_panel` (This is based on the Material Design standard-page.html).

- import the routing module
- import all the forms that may be added to the `content_panel`
- add the decorator: `@routing.template(path, priority, condition)`

```
from anvil_extras import routing
from .Form1 import Form1
from .Form2 import Form2
from .Form3 import Form3
from .ErrorForm import ErrorForm

@routing.template(path="", priority=0, condition=None)
class MainRouter(MainRouterTemplate):
```

An Anvil app can have multiple template forms. When the `url_hash` changes the routing module will check each registered template form in order of priority (highest values first). A template form will be loaded as the `open_form` only if, the current `url_hash` starts with the template's path argument **and** either the condition is `None` **or** the condition is a callable that returns `True`. The path argument can be a string or an iterable of strings.

The above example would be the fallback template form. This is equivalent to:

```
@routing.default_template
class MainRouter(MainRouterTemplate):
```

If you have a different top-level template for the `admin` section of your app you might want a second template.

```
from .. import Globals

@routing.template(path="admin", priority=1, condition=lambda: Globals.admin is not None)
class AdminRouterForm(AdminRouterTemplate):
```

The above code takes advantage of an implied `Globals` module that has an `admin` attribute. If the `url_hash` starts with `admin` and the `Globals.admin` is not `None` then this template will become the `open_form`.

Another example might be a login template

```
from .. import Globals

@routing.template(path="", priority=2, condition=lambda: Globals.user is None)
class LoginRouterForm(LoginRouterTemplate):
```

Note that `TemplateForms` are never cached (unlike `RouteForms`).

### 4.11.3 Route Forms

A route form is any form that will be loaded inside a `TemplateForm`'s `content_panel`.

- Import the routing module
- add the `@routing.route` decorator above the class definition
- The first argument to the decorator is the `url_pattern` (think of it as the page name).
- The second argument is optional and is any `url_keys` (a list of strings that make up a query strings in the `url_hash`) (use `routing.ANY` to signify optionaly `url_keys`)

```
from anvil_extras import routing

@routing.route('article', url_keys=['id'])
class ArticleForm(ArticleFormTemplate):
    ...
```

Or without any `url_keys`

```
from anvil_extras import routing

@routing.route('article')
class ArticleForm(ArticleFormTemplate):
    ...
```

Or with `url_keys` where there may be other optional keys

```
from anvil_extras import routing

@routing.route('article', url_keys=["id", routing.ANY])
class ArticleForm(ArticleFormTemplate):
    ...
```

### 4.11.4 Home form

The `HomeForm` is also a `Route Form` that appears in the `content_panel` of the loaded `TemplateForm`.

- Import the routing module
- add the `@routing.route` decorator
- set the `url_pattern` (page name) to an empty string

```
from anvil_extras import routing

@routing.route('')
class Home(HomeTemplate):
    ...
```

---

### 4.11.5 Error form (Optional)

This is the form that is shown when the `url_hash` refers to a page that does not exist, or the query string does not match the `url_keys` listed in the decorator. Follow these steps to create an error form that shows an error message:

- Create a form with the label `Sorry, this page does not exist`
- Import the routing module
- add the decorator `@routing.error_form`

```
from anvil_extras import routing

@routing.error_form
class ErrorForm(ErrorFormTemplate):
    ...
```

---

### 4.11.6 Startup Forms and Startup Modules

If you are using a Startup Module or a Startup Form all the `TemplateForms` and `RouteForms` must be imported otherwise they will not be registered by the `routing` module.

If using a Startup module, it is recommended call `routing.launch()` after any initial app logic

```
from anvil_extras import routing
from .. import Global


# Setup some global data
Global.user = anvil.server.call("get_user")
if Global.user is None:
    routing.set_url_hash("login", replace_current_url=True)

routing.launch() # I will load the correct template form
```

It is also ok to use `anvil.open_form("LoginForm")`, or to use a `TemplateForm` as the Startup Form. In either case, the `routing` module will validate the template form is correct based on the registered templates for the app.

### 4.11.7 Navigation

It is important to never use the typical method to navigate when using the `routing` module.

```
# Banned
get_open_form().content_panel.clear()
get_open_form().content_panel.add_component(Form1())
# This will result in an Exception('Form1 is a route form and was not loaded from routing
↪')
```

Instead

```
# option 1
set_url_hash('articles') # anvil's built-in method

# or an empty string to navigate to the home page
set_url_hash('')

# option 2
routing.set_url_hash('articles')
#routing.set_url_hash() method has some bonus features and is recommended over the anvil
↪'s built-in method
```

With query string parameters:

```
# option 1
set_url_hash(f'article?id={self.item["id"]}')

# option 2
routing.set_url_hash(f'article?id={self.item["id"]}')

# option 3
routing.set_url_hash(url_pattern='article', url_dict={'id':self.item['id']})
```

`routing.set_url_hash()` - has some additional features. See *API Docs* and Examples.

### 4.11.8 Dynamic Vars

An alternative to a query string is to include a dynamic URL hash. The dynamic variables inside the URL pattern will be included in the `dynamic_vars` attribute.

```
from anvil_extras import routing

@routing.route("article/{id}")
class ArticleForm(ArticleFormTemplate):
    ...
```

You can then check the `id` using:

```
print(self.dynamic_vars) # {'id': 3}
print(self.dynamic_vars['id']) # 3
```

Multiple dynanamic variables are supported e.g. `foo/{var_name_1}/{var_name_2}`. A dynamic varaible must be entirely contained within a `/` portion of the `url_pattern`, e.g. `foo/article-{id}` is not valid.

### 4.11.9 Redirects

A redirect is similar to a template in that the arguments are the same.

```
@routing.redirect(path="admin", priority=20, condition: Globals.user is None or not␣
↪Globals.user["admin"])
def redirect_no_admin():
    # not an admin or not logged in
    return "login"

# can also use routing.set_url_hash() to redirect
@routing.redirect(path="admin", priority=20, condition=lambda: Globals.user is None or␣
↪not Globals.user["admin"])
def redirect_no_admin():
    routing.set_url_hash("login", replace_current_url=True, set_in_history=False,␣
↪redirect=True)
```

When used as a decorator, the redirect function will be called if:

- the current `url_hash` starts with the redirect `path`, and
- the condition returns `True` or the condition is `None`

The redirect function can return a `url_hash`, which will then trigger a redirect. Alternatively, a redirect can use `routing.set_url_hash()` to redirect.

Redirects are checked at the same time as templates, in this way a redirect can intercept the current navigation before any templates are loaded.

### 4.11.10 API

#### Decorators

`routing.`**`template`**(*path=''*, *priority=0*, *condition=None*, *redirect=None*)

Apply this decorator above the top-level Form - `TemplateForm`.

- `path` should be a string or iterable of strings.
- `priority` should be an integer.
- `condition` can be `None`, or a function that returns `True` or `False`

The `TemplateForm` must have a `content_panel`. It is often could to refer to `TemplateForm``s with the suffix ``Router` e.g. `MainRouter`, `AdminRotuer`. There are two callbacks available to a `TemplateForm`.

**`on_navigation`**(*self*, ***nav_args*)

**`on_navigation`**(*self*, *url_hash*, *url_patter*, *url_dict*, *unload_form*)

The `on_navigation` method, when added to your `TemplateForm`, will be called whenever the `url_hash` is changed. It's a good place to adjust the look of your `TemplateForm` if the `url_hash` changes. e.g. the selected link in the sidebar. The `unload_form` is possible `None` if this is the first load of the app.

**`on_form_load`**(*self*, ***nav_args*)

**on_form_load**(*self*, *url_hash*, *url_patter*, *url_dict*, *form*)

The `on_form_load` is called after a form has been loaded into the `content_panel`. This is also a good time to adjust the `TemplateForm`.

`routing.`**default_template**

equivalent to `routing.template(path='', priority=0, condition=None)`.

`routing.`**route**(*url_pattern*, *url_keys=[]*, *title=None*, *full_width_row=False*, *template=None*)

The `routing.route` decorator should be called with arguments that determine the shape of the `url_hash`. The `url_pattern` determines the string immediately after the `#`. The `url_keys` determine the required query string parameters in a `url_hash`.

The `template`, when set, should be set to a string or list of strings that represent valid templates this route can be added to. If no `template` is set then this form can be added to any template.

The routing module adds certain parameters to a `Route Form` and supports a `before_unload` callback.

**url_hash**

The current `url_hash`. The `url_hash` includes the query. See *Introduction* for examples.

**url_pattern**

The `url_hash` without the query string.

**url_dict**

The query string is converted to a python dict.

**dynamic_vars**

See *Dynamic URLs*.

**before_unload**(*self*)

If the `before_unload` method is added it will be called whenever the form currently in the `content_panel` is about to be removed. If any truthy value is returned then unloading will be prevented. See *Form Unloading*.

`routing.`**redirect**(*path*, *priority=0*, *condition=None*)

The redirect decorator can decorate a function that will intercept the current navigtation, depending on its `path`, `priority` and `condition` arguments.

- `path` can be a string or iterable of strings.
- `priority` should be an integer - the higher the value the higher the priority.
- `conditon` should be `None` or a callable that returns a `True` or `False`.

A redirect function can return a `url_hash` - which will trigger a redirect, or it can call `routing.set_url_hash()`.

`routing.`**error_form**

The `routing.error_form` decorator is optional and can be added above a form that will be displayed if the `url_hash` does not refer to any known `Route Form`.

### Exception

**exception** routing.**NavigationExit**

Usually called inside the on_navigation callback. Prevents the current navigation from attempting to change the content_panel. Useful for login forms.

### List of Methods

routing.**launch**()

This can be called inside a Startup Module. It will ensure that the correct Template is loaded based on the current url_hash and template conditions. Calling open_form() on a TemplateForm will implicitly call routing.launch(). Until routing.launch() is called anvil components will not be loaded when the url_hash is changed. This allows you to set the url_hash in startup logic before any navigation is attempted. Similarly when a TemplateForm is loaded any routing is delayed until after the TemplateForm has been initialized.

routing.**set_url_hash**(*url_hash*)

routing.**set_url_hash**(*url_hash*, ***properties*)

routing.**set_url_hash**(*url_pattern=None*, *url_dict=None*, ***properties*)

routing.**set_url_hash**(*url_hash*, ***, *replace_current_url=False*, *set_in_history=True*, *redirect=True*, *load_from_cache=True*, ***properties*)

Sets the url_hash and begins navigation to load a form. Any properties provided will be passed to the form's properties. You can also pass the url_pattern and url_dict separately and let the routing module convert this to a valid url_hash. This is particularly useful when you have strings that need encoding as part of the query string.

The additional keywords in the call signature will adjust the routing behaviour.

If replace_current_url is set to True. Then the navigation will happen "in place" rather than as a new history item.

If set_in_history is set to False the URL will not be added to the browser's history stack.

If redirect is set to False then you do not want to navigate away from the current form.

if load_from_cache is set to False then the new URL will **not** load from cache.

Note that any additional properties will only be passed to a form if it is the first time the form has loaded and/or it is **not** loaded from cache.

routing.**get_url_components**(*url_hash=None*)

Returns a 3 tuple of the url_hash, url_pattern and url_dict. If the url_hash is None it will return the components based on the current url_hash of the page.

routing.**get_url_hash**(*url_hash=None*)

Returns the url_hash - this differs slightly from the Anvil implementation. It does not convert a query string to a dictionary automatically.

routing.**get_url_pattern**(*url_hash=None*)

Returns the part of the url_hash without the query string.

routing.**get_url_dict**(*url_hash=None*)

Returns a dictionary based on the query string of the url_hash.

routing.**load_error_form**()

Loads the error form at the current url_hash.

routing.**remove_from_cache**(*url_hash*)

Removes a `url_hash` from the `routing` module's cache.

routing.**add_to_cache**(*url_hash*, *form*)

Adds a form to the cache at a specific `url_hash`. Whenever the user navigates to this URL the cached form will be used. (Caching generally happens without you thinking about it).

routing.**clear_cache**()

Clears all forms and url_hash's from the cache.

routing.**get_cache**()

Returns the cache object from the `routing` module. Adjusting the cache directly may have side effects and is not supported.

routing.**go**(*x=0*)

Go forward/back x number of pages. Use negative values to go back.

routing.**go_back**()

Go back one page.

routing.**reload_page**(*hard=False*)

Reload the current route_form (if `hard = True` the page will refresh)

routing.**on_session_expired**(*reload_hash=True*, *allow_cancel=True*)

Override the default behaviour for a session expired. Anvil's default behaviour will reload the app at the home form.

routing.**set_warning_before_app_unload**(*True*)

Pop up the default browser dialogue when navigating away from the app.

routing.**logger**

Logging information is provided when debugging. Logging is turned off by default.

To turn logging on do: `routing.logger.debug = True`.

### 4.11.11 Notes and Examples

The following represents some notes and examples that might be helpful

#### Routing Debug Print Statements

To debug your routing behaviour use the routing logger. Routing logs are turned off by default.

To use the routing logger, in your Startup Module

```
from anvil_extras import routing

routing.logger.debug = True
```

### Page Titles

You can set each `Route Form` to have a `title` parameter, which will change the browser tab title

If you do not provide a title then the page title will be the default title provided by Anvil in your titles and logos

```
@routing.route('', title='Home | RoutingExample')
class Home(HomeTemplate):
    ...
```

```
@routing.route('article', url_keys=['id'], title="Article-{id} | RoutingExample")
class ArticleForm(ArticleFormTemplate):
    ...
```

```
@routing.route('article/{id}', title='Article | {id}')
class ArticleForm(ArticleFormTemplate):
    ...
```

- Think f-strings without the f
- Anything in curly braces should be an item from `url_keys` or a dynamic variable in the `url_pattern`.

You can also dynamically set the page title, for example, to values loaded from the database.

```
from anvil.js.window import document

@routing.route('article', url_keys=['id'])
class ArticleForm(ArticleFormTemplate):
  def __init__(self, **properties):
    self.item = anvil.server.call('get_article', article_id=self.url_dict['id'])
    document.title = f"{self.item['title']} | RoutingExample'"

    self.init_components(**properties)
```

### Full-Width Rows

You can set a `Route Form` to load as a `full_width_row` by setting the `full_width_row` parameter to `True`.

```
@routing.route('', title='Home', full_width_row=True)
class Home(HomeTemplate):
    ...
```

### Multiple Route Decorators

It is possible to define optional parameters by adding multiple decorators, e.g. one with and one without the key. Here is an example that allows using the `home page` with the default empty string and with one optional `search` parameter:

```
@routing.route('')
@routing.route('', url_keys=['search'])
class Form1(Form1Template):
  def __init__(self, **properties):
    self.init_components(**properties)
    self.search_terms.text = self.url_dict.get('search', '')
```

Perhaps your form displays a different `item` depending on the `url_pattern`/ `url_hash`:

```
@routing.route('articles')
@routing.route('blogposts')
class ListItems(ListItemsTemplate):
  def __init__(self, **properties):
    self.init_components(**properties)
    self.item = anvil.server.call(f'get_{self.url_pattern}')
    # self.url_pattern is provided by the routing module
```

### Setting a Route's Template

```
@routing.route('foo', template="MainRouter")
class Foo(FooTemplate):
    def __init__(self, **properties):
        ...
```

Setting a template argument determines which templates a route form can be added to. If no template is set then this route can be added to any template.

A template argument should be the name of the template or a list of template names.

```
@routing.route('foo', template=["MainRouter", "AdminRouter"])
class Foo(FooTemplate):
    def __init__(self, **properties):
        ...
```

If you have a route that can be used on multiple templates, consider using / notation.

```
@routing.template('admin', priority=2, condition=lambda Globals.is_admin)
class AdminRouter(AdminRouterTemplate):
    ...

@routing.route('/foo', template="AdminRouter")
class Foo(FooTemplate):
    ...
```

In the above example, since the route `"/foo"` does not start with `admin`, `"admin/foo"` will be a valid `url_pattern` for this route

This allows you to write a route for different templates and only specify the suffix.

```
@routing.template('admin', priority=2, condition=lambda Globals.is_admin)
class AdminRouter(AdminRouterTemplate):

@routing.template('accounts')
class AccountRouter(AccountRouterTemplate):

@routing.route('/foo', template=["AdminRouter", "AccountRouter"])
class Foo(FooTemplate):
```

The Foo route will be added for the url_patterns `"admin/foo"` and `"accounts/foo"`.

Note that the cached version of the Foo form will be added to either templates. If you don't want to use a cached version for different templates, you should use multiple decorators

```
@routing.route('/foo', template="AdminRouter")
@routing.route('/foo', template="AccountRouter")
class Foo(FooTemplate):
```

### Form Arguments

It's usually better to avoid required named arguments for a Form. Something like this is not allowed:

```
@routing.route('form1', url_keys=['key1'])
class Form1(Form1Template):
  def __init__(self, key1, **properties):
    ...
```

All the parameters listed in `url_keys` are required, and the rule is enforced by the routing module. If the `Route Form` has required `url_keys` then the routing module will provide a `url_dict` with the parameters from the `url_hash`.

This is the correct way:

```
@routing.route('form1', url_keys=['key1'])
class Form1(Form1Template):
  def __init__(self, **properties):
    key1 = self.url_dict['key1']
    #routing provides self.url_dict
```

If you need a catch all for arbirtrary url_keys use `url_keys=[routing.ANY]`. Or combine `routing.ANY` with required keys `url_keys=["search", routing.ANY]`.

### Template Form Callbacks

There are two callbacks available for a `TemplateForm`.

- `on_navigation`: called whenever the `url_hash` changes
- `on_form_load`: called after a form is loaded into the `content_panel`

#### `on_navigation` example:

To use the Material Design role `'selected'` for sidebar links, create an `on_navigation` method in your `TemplateForm`.

```
@routing.default_template
class MainForm(MainFormTemplate):
  def __init__(self, **properties):
    self.init_components(**properties)
    self.links = [self.articles_link, self.blog_posts_link]
    self.blog_posts_link.tag.url_hash = 'blog-posts'
    self.articles_link.tag.url_hash   = 'articles'

  def on_navigation(self, **nav_args):
    # this method is called whenever routing provides navigation behaviour
    # url_hash, url_pattern, url_dict are provided by the template class decorator
    for link in self.links:
```

(continues on next page)

(continued from previous page)

```
      if link.tag.url_hash == nav_args.get('url_hash'):
        link.role = 'selected'
      else:
        link.role = 'default'
```

**Nav Args will look like:**

```
nav_args = {'url_hash':    url_hash,
            'url_pattern': url_pattern,
            'url_dict':    url_dict,
            'unload_form': form_that_will_be_unloaded # could be None if initial call
            }
```

#### on_form_load example:

If you want to use animation when a form is loaded you might use the on_form_load method.

```
def on_form_load(self, **nav_args):
    # this method is called whenever the routing module has loaded a form into the␣
↪content_panel
    form = nav_args["form"]
    animate(form, fade_in, duration=300)
```

Note if you wanted to use a fade-out you could also use the on_navigation method.

```
def on_navigation(self, **nav_args):
    # this method is called whenever the url_hash changes
    form = nav_args["unload_form"]
    if form is not None:
        animate(form, fade_out, duration=300).wait()
        # wait for animation before continuing
```

### Navigation Techniques

#### redirect=False

It is possible to set a new URL without navigating away from the current form. For example, a form could have this code:

```
def search_click(self, **event_args):
  if self.search_terms.text:
      routing.set_url_hash(f'?search={self.search_terms.text}',
                           redirect=False
                          )
  else:
      routing.set_url_hash('',
                           redirect=False,
                          )
  self.search(self.search_terms.text)
```

This way search parameters are added to the history stack so that the user can navigate back and forward, but routing does not attempt to navigate to a new form instance.

### Important

Be careful if you use `routing.set_url_hash` inside the `__init__` method or `form_show` event. You may cause an infinite loop if your `url_hash` points to the same form and `redirect=True`! In this case, you will get a `warning` from the `routing.logger` and navigation/redirection will be halted.

Navigation will be halted after 5 navigation attempts without loading a form to the `content_panel`.

#### `replace_current_url=True`

It is also possible to replace the current URL in the history stack rather than creating a new entry in the history stack.

In the demo app the `ArticleForm` creates a new article if the `id` parameter is empty like: `url_hash = "article?id="`

```
@routing.route('article', url_keys=['id'])
class ArticleForm(ArticleFormTemplate):
    def __init__(self, **properties):
        self.init_components(**properties)
        if url_dict['id']:
            self.item = anvil.server.call("get_article_by_id", self.url_dict['id'])
        else:
            # url_dict['id'] is empty
            self.item = anvil.server.call('create_new_article')
            routing.set_url_hash(f"article?id={self.item['id']}",
                                 replace_current_url=True,
                                 set_in_history=True,
                                 redirect=False
                                )
```

See *API Docs* for a list of valid kwargs for `routing.set_url_hash()`.

### Security

**Security issue**: You log in, open a form with some data, go to the next form, log out, go back 3 steps and you see the cached stuff that was there when you were logged in.

**Solution 1**: When a form shows sensitive data it should always check for user permission in the `form_show` event, which is triggered when a cached form is shown.

**Solution 2**: Call `routing.clear_cache()` to remove the cache upon logging out.

---

### Preventing a Form from Unloading (when navigating within the app)

Create a method in a `Route Form` called `before_unload`

To prevent Unloading return a value

```
def before_unload(self):
  # this method is called when the form is about to be unloaded from the content_panel
  if confirm('are you sure you want to close this form?'):
    pass
  else:
    return 'STOP'
```

*NB*: - Only use if you need to prevent unloading. - Otherwise, the `form_hide` event should work just fine.

*NB*: - This method does not prevent a user from navigating away from the app entirely. (see the section *Leaving the App* below)

---

### Passing properties to a form

You can pass properties to a form by adding them as keyword arguments to `routing.set_url_hash`

```
def article_link_click(self, **event_args):
    routing.set_url_hash(f'article?id={self.item["id"]}', item=self.item)
```

---

### I have a login form how do I work that?

**As part of anvil_extras.routing**

Login forms are the default form to load if no user is logged in.

You could create a login template. We don't want the user to navigate back/forward to other `routes` within our app once the user has logged out.

You can avoid this by raising a `routing.NavigationExit()` exception in the `on_navigation()` callback.

```
@routing.template("", priority=10, condition=lambda: Globals.user is None)
class LoginForm(LoginFormTemplate):
    def on_navigation(self, **url_args):
        raise routing.NavigationExit()
        # prevent routing from changing the content panel based on the hash if the user␣
↪tries to navigate back to a previous page

    def login_button_click(self, **event_args):
        user = anvil.users.login_with_form()
        if user is not None:
            Globals.user = user
            routing.set_url_hash("")
```

You may choose to use redirect functions to intercept the navigation.

```
@routing.redirect("", priority=10, condition=lambda: Globals.user is None)
def redirect():
    return "login"

@routing.redirect("login", priority=10, condition=lambda: Globals.user is not None)
def redirect():
    # we're logged in - don't go to the login form
    return ""

@routing.default_template
class DashboardRouter(DashboardRouterTemplate):
    ...

@routing.template("login", priority=1)
class LoginRouter(LoginRouterTemplate):
    def on_navigation(self, url_hash, **url_args):
        raise routing.NavigationExit
        # prevent routing from changing the content panel

    def login_button_click(self, **event_args):
        Globals.user = anvil.users.login_with_form()
        routing.set_url_hash("", replace_current_url=True)
        # let routing decide which template
```

Advanced - redirect back to the url hash that was being accessed

```
@routing.redirect("", priority=10, condition=lambda: Globals.user is None)
def redirect():
    current_hash = routing.get_url_hash()
    routing.set_url_hash("login", current_hash=current_hash, replace_current_url=True,␣
↪set_in_history=False)
    # the extra property current_hash passed to the form as a keyword argument

@routing.redirect("login", priority=10, condition=lambda: Globals.user is not None)
def redirect():
    # we're logged in - don't go to the login form
    return ""

@routing.default_template
class DashboardRouter(DashboardRouterTemplate):
    ...

@routing.template("login", priority=1)
class LoginRouter(LoginRouterTemplate):
    def __init__(self, current_hash="", **properties):
        self.current = current_hash

    def on_navigation(self, url_hash, **url_args):
        self.current = url_hash
        routing.set_url_hash("login", replace_current_url=True, set_in_history=False,␣
↪redirect=False)
        raise routing.NavigationExit
        # prevent routing from changing the content panel
```

(continues on next page)

(continued from previous page)

```
    def login_button_click(self, **event_args):
        Globals.user = anvil.users.login_with_form()
        routing.set_url_hash(self.current, replace_current_url=True)
        # let routing decide which template to load
```

More advanced - to access the current `url_hash` that is stored in the browser's history you can use `window.history.state.get.url`.

```
@routing.redirect("", priority=10, condition=lambda: Globals.user is None)
def redirect():
    return "login"

@routing.redirect("login", priority=10, condition=lambda: Globals.user is not None)
def redirect():
    return ""

@routing.default_template
class DashboardRouter(DashboardRouterTemplate):
    ...

@routing.template("login", priority=1)
class LoginRouter(LoginRouterTemplate):
    def on_navigation(self, **url_args):
        routing.set_url_hash("login", replace_current_url=True, set_in_history=False,␣
↪redirect=False)
        raise routing.NavigationExit
        # prevent routing from changing the content panel

    def login_button_click(self, **event_args):
        Globals.user = anvil.users.login_with_form()
        from anvil.js.window import history
        routing.set_url_hash(history.state.url, replace_current_url=True)
```

Alternatively, you could load the login form as a `route` form rather than a template.

```
@routing.default_template
class MainRouter(MainRouterTemplate):
    def __init__(self, **properties):
        if Globals.users is None:
            routing.set_url_hash("login") # this logic could also be in a Startup Module

    def on_navigation(self, url_hash, **url_args):
        if Globals.user is None and url_hash != "login":
            raise routing.NavigationExit()
            # prevent routing from changing the login route form inside the content panel


@routing.route('login')
class LoginForm(LoginFormTemplate):
    def __init__(self, **properties):
        self.init_components(**properties)
```

(continues on next page)

(continued from previous page)

```
  def form_show(self, **event_args):
    """This method is called when the column panel is shown on the screen"""
    user = anvil.users.get_user()
    while not user:
      user = anvil.users.login_with_form()

    routing.remove_from_cache(self.url_hash)  # prevents the login form loading from␣
↪cache in the future...
    routing.set_url_hash('',
                         replace_current_url=True,
                         redirect=True
                         )
    # " replaces 'login' in the history stack and redirects to the HomeForm
```

**Separate from anvil_extras.routing**

Rather than have the `LoginForm` be part of the navigation, you could create a startup module that will call `open_form("LoginForm")` if no user is logged in. The `LoginForm` should **not** have any `anvil_extras.routing` decorators.

Then when the user has signed in you can call `open_form('MainForm')`. The `routing` module will return to changing `templates` and load `routes` when the `url_hash` changes.

When the user signs out you can call `open_form('LoginForm')`. `routing` will no longer take control of the navigation. There will still be entries when the user hits back/forward navigation (i.e. the `url_hash` will change but there will be no change in forms...) :smile:

It is a good idea to call `routing.clear_cache()` when a user logs out.

---

### I have a page that is deleted - how do I remove it from the cache?

```
def trash_link_click(self, **event_args):
  """called when trash_link is clicked removes the """
  self.item.delete()  # table row
  routing.remove_from_cache(self.url_hash) # self.url_hash provided by the @routing.
↪route class decorator
  routing.set_url_hash('articles',
                       replace_current_url=True,
                     )
```

And in the `__init__` method - you will want something like:

```
@routing.route('article', keys=['id'], title='Article-{id}')
class ArticleForm(ArticleFormTemplate):
  def __init__(self, **properties):
    try:
      self.item = anvil.server.call('get_article_by_id', self.url_dict['id'])
    except:
      routing.set_url_hash('articles', replace_current_url=True)
      raise Exception('This article does not exist or has been deleted')
```

---

### Form Show is important

since the forms are loaded from cache you may want to use the `form_show` events if there is a state change

### Example 1

When that article was deleted in the above example we wouldn't want the deleted article to show up on the `repeating_panel`

so perhaps:

```
@routing.route('articles')
class ListArticlesForm(ListArticlesFormTemplate):
  def __init__(self, **properties):
    # Set Form properties and Data Bindings.
    self.init_components(**properties)
    self.repeating_panel.items = anvil.server.call('get_articles')

    # Any code you write here will run when the form opens.

  def form_show(self, **event_args):
    """This method is called when the column panel is shown on the screen"""
    self.repeating_panel.items = anvil.server.call_s('get_articles')
    # silent call to the server on form show
```

**An alternative approach to the above scenario:**

set `load_from_cache=False`

That way you wouldn't need to utilise the show event of the `ListArticlesForm`

```
@routing.route('article', keys=['id'], title='Article-{id}')
class ArticleForm(ArticleFormTemplate):
  def __init__(self, **properties):
    try:
      self.item = anvil.server.call('get_article_by_id', self.url_dict['id'])
    except:
      routing.set_url_hash('articles', replace_current_url=True, load_from_cache=False)

  def trash_link_click(self, **event_args):
    """called when trash_link is clicked removes the """
    self.item.delete()  # table row
    routing.remove_from_cache(self.url_hash) # self.url_hash provided by the @routing.
↪route class decorator
    routing.set_url_hash('articles',
                         replace_current_url=True,
                         load_from_cache=False)
```

### Example 2

In the search example above the same form represents multiple `url_hash`s in the cache.

No problem.

Whenever navigation is triggered by clicking the back/forward buttons, the `self.url_hash`, `self.url_dict` and `self.url_pattern` are updated and the `form_show` event is triggered.

```
def form_show(self, **event_args):
  search_text = self.url_dict.get('search','')
  self.search_terms.text = search_text
  self.search(search_text)
```

---

### Leaving the app

Routing implements W3 Schools onbeforeunload method.

This warns the user before navigating away from the app using a default browser warning. (This may not work on ios)

By default, this setting is switched off. To switch it on do: `routing.set_warning_before_app_unload(True)`

To implement this behaviour for all pages change the setting in your Startup Module.

To implement this behaviour only on specific `Route Forms` toggle the setting like:

```
def form_show(self, **event_args):
  routing.set_warning_before_app_unload(True)

def form_hide(self, **event_args):
  routing.set_warning_before_app_unload(False)
```

Or based on a parameter (See the example app `ArticleForm` code for a working example)

```
def edit_status_toggle(status):
  routing.set_warning_before_app_unload(status)
```

*NB:* When used on a specific `Route Form` this should be used in conjunction with the `before_unload` method (see above).

## 4.12 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.

### 4.12.1 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()
```

(continues on next page)

(continued from previous page)

```
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()
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)
```

(continues on next page)

(continued from previous page)

```
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"}]
```

## 4.13 Storage

### 4.13.1 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.

### 4.13.2 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:
            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 = ""
```

### 4.13.3 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 dev-tools 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.

## 4.14 Utils

Client and server-side utility functions.

### 4.14.1 import_module

Very similar to python's `importlib.import_module` implementation. Use in the same way.

Takes two arguments, the `name` to import, and an optional `package`.

The 'package' argument is required when performing a relative import. It specifies the package to use as the anchor point from which to resolve the relative import to an absolute import.

**Example implementation:**

```
from anvil_extras.utils import import_module
from functools import cache

class MainForm(MainFormTemplate):
    ...

    def link_click(self, sender, **event_args):
        self.load_form(sender.tag)

    @cache
    def get_form(self, form_name):
        form_module = import_module(f".{form_name}", __package__)
        form_cls = getattr(form_module, form_name)
        return form_cls()


    def load_form(self, form_name):
        form = self.get_form(form_name)
        self.content_panel.clear()
        self.content_panel.add_component(form)
```

### 4.14.2 Timing

#### timed decorator

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

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


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

The decorator takes a `logging.Logger` instance as one of its optional keyword arguments. On both the server and the client this can be a Logger from the anvil_extras logging module. On the server, this can be from the Python `logging` module.

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

The default logger is an anvil_extras Logger instance, which will log to stdout. Messages will appear in your App's logs and the IDE console. You can, however, create your own logger and pass that instead if you need more sophisticated behaviour:

```
import logging
from anvil_extras.utils import timed

my_logger = logging.getLogger(__name__)


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

```
from anvil_extras.utils import timed, logging

my_logger = logging.Logger(name="Timing", format={"{name}: {time:%H:%M:%S}-{msg}"},␣
↪level=logging.DEBUG)

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

### 4.14.3 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)
```

The form's `item` property will be proxied.

If your original item was a dictionary, whenever a value of the proxied item changes, the form's `refresh_data_bindings` method will be called.

Note that the proxied item will make changes to the original `item`.

It shouldn't matter what the original `item` is. It could be a dictionary, app_table Row or some other obsucre object.

### 4.14.4 Wait for writeback

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

This is particularly useful if you have a form with text fields. Race conditions 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.

### 4.14.5 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)
        ...
```

## 4.15 Zod

Functional approach to data validation.
Independent of UI.
Available client and server side.
Attempts to match python typing.
Heavily based on the TypeScript library zod.dev.

### 4.15.1 Demo App

Clone Link
Live Demo

### 4.15.2 Basic Usage

Creating a simple string schema

```
from anvil_extras import zod as z

# create a schema
schema = z.string()

# parsing
schema.parse("tuna") # -> "tuna"
schema.parse(42) # -> throws ParseError

# "safe" parsing - doesn't throw if valid
result = schema.safe_parse("tuna") # -> ParseResult(success=True, data="tuna")
result.success # True
result = schema.safe_parse(42) # -> ParseResult(success=False, error=ParseError("Invalid␣
↪type"))
result.success # False
```

Creating a typed_dict schema

```
from anvil_extras import zod as z

# create a schema
user = z.typed_dict({
    "username": z.string()
})

user.parse({"username": "Meredydd"}) # -> {"username": "Meredydd"}
```

### 4.15.3 Primitives

```
from anvil_extras import zod as z

z.string()
z.integer()
z.float()
z.number() # int or float
z.boolean()
z.date()
z.datetime()
z.none()

# catch all types - allow any value
```

(continues on next page)

(continued from previous page)

```
z.any()
z.unknown()

# never types - allows no values
z.never()
```

### 4.15.4 Literals

```
from anvil_extras import zod as z

tuna = z.literal("tuna")
empty_str = z.literal("")
true = z.literal(True)
_42 = z.literal(42)

# retrieve the literal value
tuna.value # "tuna"
```

### 4.15.5 Strings

Zod includes a handful of string-specific validations.

```
z.string().max(5)
z.string().min(5)
z.string().len(5)
z.string().email()
z.string().url()
z.string().uuid()
z.string().regex(re.compile(r"^\d+$""))
z.string().startswith(string)
z.string().endswith(string)
z.string().strip() # strips whitespace
z.string().lower() # convert to lower case
z.string().upper() # convert to upper case
z.string().datetime() # defaults to iso format string
z.string().date() # defaults to iso format string
```

You can customize some common error messages when creating a string schema.

```
name = z.string(
    required_error="Name is required",
    invalid_type_error="Name must be a string",
)
```

When using validation methods, you can pass in an additional argument to provide a custom error message

```
z.string().min(5, message="Must be 5 or more characters long")
z.string().max(5, message="Must be 5 or fewer characters long")
z.string().length(5, message="Must be exactly 5 characters long")
```

(continues on next page)

(continued from previous page)

```
z.string().email(message="Invalid email address")
z.string().url(message="Invalid url")
z.string().uuid(message="Invalid UUID")
z.string().startswith("https://", message="Must provide secure URL")
z.string().endswith(".com", message="Only .com domains allowed")
z.string().datetime(message="Invalid datetime string! Must be in isoformat")
```

### 4.15.6 Coercion for primitives

Zod provides a convenient way to coerce primitive values.

```
schema = z.coerce.string()

# remove print statements
schema.parse("tuna")  # => "tuna"
schema.parse(12)      # => "12"
schema.parse(True)    # => "True"
```

During the parsing step, the input is passed through the `str()` function. Note that the returned schema is a ZodString instance so you can use all string methods.

```
z.coerce.string().email().min(5)
```

The following primitive types support coercion

```
z.coerce.string() # str(input)
z.coerce.boolean() # bool(input)
z.coerce.integer() # int(input)
z.coerce.float() # float(input)
```

The int and float coercions will be surrounded in a try/except. This way coercion failures will be reported as invalid type errors.

### 4.15.7 Numbers, Integers and Floats

Zod integer and float expect their equivalent python types when parsed. A zod number accepts either integer or float.

```
from anvil_extras.zod import z

age = z.number(
    required_error="Age is required",
    invalid_type_error="Age must be a number",
)
```

Zod includes a handful of number-specific validations.

```
from anvil_extras.zod import z

z.number().gt(5)
z.number().ge(5)  # greater than or equal to, alias .min(5)
z.number().lt(5)
```

(continues on next page)

(continued from previous page)

```
z.number().le(5)  # less than or equal to, alias .max(5)

z.number().int()  # value must be an integer

z.number().positive()     # > 0
z.number().nonnegative()  # >= 0
z.number().negative()     # < 0
z.number().nonpositive()  # <= 0
```

The equivalent validations are available on `integer` and `float`.

Optionally, you can pass in a second argument to provide a custom error message.

```
z.number().le(5, message="thisistoobig")
```

### 4.15.8 Booleans

You can customize certain error messages when creating a boolean schema

```
is_active = z.boolean(
    required_error="isActive is required",
    invalid_type_error="isActive must be a boolean",
)
```

### 4.15.9 Dates and Datetimes

```
from anvil_extras.zod import z
from datetime import date

z.date().safe_parse(date.today())  # success: True
z.date().safe_parse("2022-01-12")  # success: False
```

You can customize the error messages

```
my_date_schema = z.date(
    required_error="Please select a date and time",
    invalid_type_error="That's not a date!",
)
```

Zod provides a handful of datetime-specific validations.

```
z.date().min(
    date(1900, 1, 1),
    message="Too old"
)
z.date().max(
    date.today(),
    message="Too young!"
)
```

#### Supporting date strings

```
def preprocess_date(arg):
    if isinstance(arg, str):
        try:
            return date.fromisoformat(arg) #could use datetime.strptime().date
        except Exception:
            return arg

    else:
        return arg

date_schema = z.preprocess(preprocess_date, z.date())

date_schema.safe_parse(date(2022, 1, 12))  # success: True
date_schema.safe_parse("2022-01-12")  # success: True
```

### 4.15.10 Enums

```
from anvil_extras.zod import z

FishEnum = z.enum(["Salmon", "Tuna", "Trout"])
```

`z.enum` is a way to declare a schema with a fixed set of allowable values. Pass the list of values directly into `z.enum()`.

To retrieve the enum options use `.options`

```
FishEnum.options  # ["Salmon", "Tuna", "Trout
```

### 4.15.11 Optional

Optional is synonymous with python's typing.Optional. In other words, something optional can also be None. (This is different to Zod TypeScript's `optional`)

```
from anvil_extras.zod import z

schema = z.optional(z.string())

schema.parse(None)  # returns None
```

For convenience, you can also call the `.optional()` method on an existing schema.

```
schema = z.string().optional()
```

You can extract the wrapped schema from a ZodOptional instance with `.unwrap()`.

```
string_schema = z.string()
optional_string = string_schema.optional()
optional_string.unwrap() == string_schema # True
```

### 4.15.12 TypedDict

This is equivalent to Zod TypeScript's `object` schema. We chose `typed_dict` since it matches Python's `typing.TypedDict`. (`z.object` is also available for convenience)

```
from anvil_extras.zod import z

# all properties are required by default
Dog = z.typed_dict({
    "name": z.string(),
    "age": z.number()
})
```

#### API

**class ZodTypedDict**

**shape**

Use `.shape` to access the schemas for a particular key.

```
Dog.shape["name"]  # => string schema
Dog.shape["age"]   # => number schema
```

**keyof**()

Use `.keyof` to create a ZodEnum schema from the keys of a typed_dict schema.

```
key_schema = Dog.keyof()
key_schema # ZodEnum<["name", "age"]>
```

**extend**()

You can add additional fields to a typed_dict schema with the .extend method.

```
from anvil_extras.zod import z

# all properties are required by default
Dog = z.typed_dict({
    "name": z.string(),
    "age": z.number()
})

DogWithBreed = Dog.extend({
    "breed": z.string()
})
```

You can use `.extend` to overwrite fields! Be careful with this power!

**merge**(*B*)

Equivalent to `A.extend(B.shape)`.

If the two schemas share keys, the properties of B overrides the property of A. The returned schema also inherits the "unknownKeys" policy (strip/strict/passthrough) and the catchall schema of B.

```
BaseTeacher = z.typed_dict({
    "students": z.list(z.string())
})

HasID = z.typed_dict({
    "id": z.string()
})

Teacher = BaseTeacher.merge(HasID)

# the type of the `Teacher` variable is inferred as follows:
# {
#     "students": z.array(z.string()),
#     "id": z.string()
# }
```

**pick**(*keys=None*)

Returns a modified version of the typed_dict schema that only includes the keys specified in the `keys` argument. (This method is inspired by TypeScript's built-in `Pick` utility type).

```
from anvil_extras.zod import z

Recipe = z.typed_dict({
    "id": z.string(),
    "name": z.string(),
    "ingredients": z.list(z.string()),
})

JustTheName = Recipe.pick(["name"])

# the type of the JustTheName variable is inferred as follows:
# {
#     "name": z.string()
# }
```

**omit**(*keys=None*)

Returns a modified version of the typed_dict schema that excludes the keys specified in the `keys` argument. (This method is inspired by TypeScript's built-in `Omit` utility type).

```
from anvil_extras.zod import z

Recipe = z.typed_dict({
    "id": z.string(),
    "name": z.string(),
    "ingredients": z.list(z.string()),
})

NoIDRecipe = Recipe.omit(["id"])

# the type of the `NoIDRecipe` variable is inferred as follows:
# {
#     "name": z.string(),
#     "ingredients": z.list(z.string())
```

(continues on next page)

(continued from previous page)

```
# }
```

**partial**(*keys=None*)

**Returns**
a modified version of the typed_dict schema in which all properties are made optional. This method is inspired by the built-in TypeScript utility type *Partial*.

**Parameters**
**keys** (*iterable*) – Optional argument that specifies which properties to make optional. If not provided, all properties are made optional.

```
from anvil_extras.zod import z

User = z.typed_dict({
    "email": z.string(),
    "username": z.string(),
})

# create a partial version of the `User` schema
PartialUser = User.partial()

PartialUser.parse({"email": "foo@gmail.com"}) # -> {"email": "foo@gmail.com"}
PartialUser.parse({}) # -> {}
PartialUser.parse({"email": None}) # -> raises ParseError
```

the type of the *PartialUser* variable is equivalent to:

```
{
    "email": z.string().not_required(),
    "username": z.string().not_required(),
}
```

In other words the parsed dictionary may or may not include the `"email"` and `"username"` key. Note this is different to `.optional()` which would allow the value to be None

Create a partial version of the *User* schema where only the *email* property is made optional

```
OptionalEmail = User.partial(["email"])

# the type of the `OptionalEmail` variable is equivalent to:
# {
#     "email": z.string().not_required(),
#     "username": z.string(),
# }
```

**required**(*keys=None*)

Returns a modified version of the typed_dict schema in which all properties are made required. This method is the opposite of the `.partial` method, which makes all properties optional.

**Parameters**
**keys** (*iterable*) – Optional argument that specifies which properties to make required. If not provided, all properties are made required.

```
from anvil_extras.zod import z

User = z.typed_dict({
    "email": z.string(),
    "username": z.string(),
}).partial()

# create a required version of the `User` schema
RequiredUser = User.required()
```

`RequiredUser` is now equivalent to the original shape.

Create a required version of the `User` schema where only the `email` property is made required

```
RequiredEmail = User.required(["email"])

# the type of the `RequiredEmail` variable is equivalent to:
# {
#     "email": z.string(),
#     "username": z.string().not_required(),
# }
```

**passthrough**()

Returns a modified version of the typed_dict schema that enables "passthrough" mode. In passthrough mode, unrecognized keys are not stripped out during parsing.

```
from anvil_extras.zod import z

Person = z.typed_dict({
    "name": z.string(),
})

# parse a dict with unrecognized keys
result = Person.parse({
    "name": "bob dylan",
    "extraKey": 61,
})

# the `result` variable is as follows:
# {
#     "name": "bob dylan",
# }
```

The `extraKey` property has been stripped out because the `Person` schema is not in "passthrough" mode

```
# enable "passthrough" mode for the `Person` schema
PassthroughPerson = Person.passthrough()

# parse a dict with unrecognized keys
result = PassthroughPerson.parse({
    "name": "bob dylan",
    "extraKey": 61,
})
```

(continues on next page)

(continued from previous page)

```
# the `result` variable is now as follows:
# {
#     "name": "bob dylan",
#     "extraKey": 61,
# }
```

Now the `extraKey` property has not been stripped out because the `PassthroughPerson` schema is in `"passthrough"` mode

**strict**()

Returns a modified version of the typed_dict schema that disallows unknown keys during parsing. If the input to `.parse()` contains any unknown keys, a `ParseError` will be thrown.

```
from anvil_extras.zod import z

Person = z.typed_dict({
    "name": z.string(),
})

# parse a dict with unrecognized keys
try:
    result = Person.strict().parse({
        "name": "bob dylan",
        "extraKey": 61,
    })
except z.ParseError as e:
    print(e)
    # => "Unknown key 'extraKey' found in input at path 'extraKey'"
```

The code above will throw a ParseError because the `Person` schema is in `"strict"` mode and the input contains an unknown key

**strip**()

Returns a modified version of the typed_dict schema that strips out unrecognized keys during parsing. This is the default behavior of ZodTypedDict schemas.

**catchall**(*schema: ZodAny*) → *ZodTypedDict*

You can pass a `"catchall"` schema into a typed_dict schema. All unknown keys will be validated against it.

**Parameters**
**schema** – A Zod schema for validating unknown keys.

**Returns**
A new ZodTypedDict schema with catchall schema for unknown keys.

**Raises**
**ParseError** – If any unknown keys fail validation.

Example:

```
from zod import z

# Create a person schema with `name` field
person = z.typed_dict({
```

(continues on next page)

(continued from previous page)

```
    "name": z.string()
})

# Add a catchall schema for any unknown keys
person = person.catchall(z.number())

# Parse with valid extra key
person.parse({
    "name": "bob dylan",
    "validExtraKey": 61
})

# Parse with invalid extra key
person.parse({
    "name": "bob dylan",
    "invalidExtraKey": "foo"
})
# => raises ParseError
```

Using `.catchall()` obviates `.passthrough()`, `.strip()`, or `.strict()`. All keys are now considered "known".

### 4.15.13 NotRequired

The `.not_required()` method can be used in conjunction with typed_dict schemas. This means the key value pair can be missing. See the *ZodTypedDict.partial()* method.

### 4.15.14 List

Similar to `typing.List` type.

```
string_list = z.list(z.string())

# equivalent
string_array = z.string().list()
```

Be careful with the `.list()` method. It returns a new ZodList instance. This means the order in which you call methods matters. For instance:

```
z.string().optional().list() # (string | None)[]
z.string().list().optional() # string[] | None
```

A ZodList schema will parse a `tuple` or `list`. A `tuple` will be returned as a `list` upon parsing.

The following method are provided on a `list` schema

```
z.string().list().min(5)  # must contain 5 or more items
z.string().list().max(5)  # must contain 5 or fewer items
z.string().list().len(5)  # must contain 5 items exactly
```

### Additional API

**class ZodList**

**element**

Use `.element` to access the schema for an element of the array.

```
string_list.element; # => string schema
```

**nonempty**(*message*)

If you want to ensure that an array contains at least one element, use `.nonempty()`.

**Parameters**
**message** – Optional custom error message.

**Returns**
The same ZodList instance with `.nonempty()` added.

Example:

```
non_empty_strings = z.string().list().nonempty();
non_empty_strings.parse([]); // throws: "List cannot be empty"
non_empty_strings.parse(["Ariana Grande"]); # passes
```

You can optionally specify a custom error message:

```
from anvil_extras import zod as z

# optional custom error message
non_empty_strings = z.string().array().nonempty(
    message="Can't be empty!"
)
```

## 4.15.15 Tuples

Unlike lists, tuples have a fixed number of elements and each element can have a different type. It is similar to `typing.Tuple` type.

```
athlete_schema = z.tuple([
    z.string(), # name
    z.integer(), # jersey number
    z.dict({"points_scored": z.number()}) # statistics
])
```

A variadic ("rest") argument can be added with the `.rest` method.

```
from anvil_extras import zod as z

variadic_tuple = z.tuple([z.string()]).rest(z.number())
result = variadic_tuple.parse(["hello", 1, 2, 3])]
```

For convenience a tuple schema will parse both A `list` and a `tuple` in the same way.

### 4.15.16 Unions

Zod includes a built-in `z.union` method for composing "OR" types. This is similar to `typing.Union`.

```
string_or_number = z.union([z.string(), z.number()])

string_or_number.parse("foo") # passes
string_or_number.parse(14) # passes
```

Zod will test the input against each of the "options" in order and return the first value that validates successfully.

For convenience, you can also use the `.union` method:

```
string_or_number = z.string().union(z.number())
```

### 4.15.17 Mappings

Mappings are similar to Python's `typing.Mapping` or `typing.Dict` types. You should specify a key and value schema

```
NumberCache = z.mapping(z.string(), z.integer());

# expects to parse dict[str, int]
```

This is particularly useful for storing or caching items by ID

```
user_schema = z.typed_dict({"name": z.string()})
user_cache_schema = z.mapping(z.string().uuid(), user_schema)

user_store = {}

user_store["77d2586b-9e8e-4ecf-8b21-ea7e0530eadd"] = {"name": "Carlotta"}
user_cache_schema.parse(user_store) # passes


user_store["77d2586b-9e8e-4ecf-8b21-ea7e0530eadd"] = {"whatever": "Ice cream sundae"}
user_cache_schema.parse(user_store) # Fails
```

### 4.15.18 Recursive types

```
from anvil_extras import zod as z

Category = z.lazy(lambda:
    z.typed_dict({
        'name': z.string(),
        'subcategories': z.list(Category),
    })
)

Category.parse({
    'name': 'People',
```

(continues on next page)

(continued from previous page)

```
    'subcategories': [
        {
        'name': 'Politicians',
        'subcategories': [{ 'name': 'Presidents', 'subcategories': [] }],
        },
    ],
}) # passes
```

If you want to validate any JSON value, you can use the snippet below.

```
literal_schema = z.union([z.string(), z.number(), z.boolean(), z.none()])
json_schema = z.lazy(lambda: z.union([literal_schema, z.list(json_schema), z.
↪mapping(json_schema)]))

json_schema.parse(data)
```

### 4.15.19 Isinstance

You can use `z.isinstance` to check that the input is an instance of a class. This is useful to validate inputs against classes.

```
from anvil_extras import zod as z

class Test:
    def __init__(self, name: str):
        self.name = name

TestSchema = z.isinstance(Test)

blob = "whatever"
TestSchema.parse(Test("my_name")) # passes
TestSchema.parse(blob) # throws
```

### 4.15.20 Preprocess

Typically Zod operates under a "parse then transform" paradigm. Zod validates the input first, then passes it through a chain of transformation functions. (For more information about transforms)

But sometimes you want to apply some transform to the input before parsing happens. A common use case: type coercion. Zod enables this with the `z.preprocess()`.

```
cast_to_string = z.preprocess(lambda val: str(val), z.string())
```

### 4.15.21 Schema Methods

**parse**(*data*)

**Returns**
If the given value is valid according to the schema, a value is returned. Otherwise, an error is thrown.

IMPORTANT: The value returned by *.parse* is a deep clone of the variable you passed in.

**Example**

```
string_schema = z.string()
string_schema.parse("fish")  # returns "fish"
string_schema.parse(12)  # throws ParseError
```

**safe_parse**(*data*)

**Returns**
`ParseResult(success: bool, data: any, error: ParseError | None)`

If you don't want Zod to throw errors when validation fails, use `.safe_parse`. This method returns a ParseResult containing either the successfully parsed data or a ParseError instance containing detailed information about the validation problems.

**Example**

```
string_schema.safe_parse(12)  # ParseResult(success=False, error=ParseError)
string_schema.safe_parse("fish")  # ParseResult(success=True, data="fish")
```

You can handle the errors conveniently:

```
result = stringSchema.safeParse("billie")
if not result.success:
    # handle error then return
    print(result.error)
else:
    # do something
    print(result.data)
```

Not Yet Documented:

- refine
- super_refine
- transform
- super_transform
- default
- catch
- optional
- error handling and formatting
- pipe

CHAPTER

FIVE

# INDICES AND TABLES

- genindex
- modindex
- search

# INDEX


## A

animate()
    built-in function, 21
Animation (*built-in class*), 21
AsyncCall (*built-in class*), 42
await_result() (*AsyncCall method*), 42

## B

before_unload(), 56
built-in function
    animate(), 21
    call_async(), 42
    cancel(), 42
    defer(), 43
    dismiss_on_outside_click(), 49
    dismiss_on_scroll(), 49
    format_selected_text(), 9
    get_bounding_rect(), 25
    has_popover(), 49
    is_animating(), 25
    repeat(), 42
    routing.add_to_cache(), 58
    routing.clear_cache(), 58
    routing.get_cache(), 58
    routing.get_url_components(), 57
    routing.get_url_dict(), 57
    routing.get_url_hash(), 57
    routing.get_url_pattern(), 57
    routing.go(), 58
    routing.go_back(), 58
    routing.launch(), 57
    routing.load_error_form(), 57
    routing.on_session_expired(), 58
    routing.redirect(), 56
    routing.reload_page(), 58
    routing.remove_from_cache(), 57
    routing.route(), 56
    routing.set_url_hash(), 57
    routing.set_warning_before_app_unload(), 58
    routing.template(), 55
    set_default_container(), 49
    set_default_max_width(), 49
    wait_for(), 25, 42

## C

call_async()
    built-in function, 42
cancel(), 26
    built-in function, 42
catchall() (*ZodTypedDict method*), 87
check(), 34
clear() (*LocalStorageWrapper method*), 74
commitStyles(), 26
create_store() (*LocalStorageWrapper class method*), 73
critical() (*Logger method*), 33
cubic_bezier(), 27

## D

debug() (*Logger method*), 33
default_template (*routing attribute*), 56
defer()
    built-in function, 43
disabled (*Logger attribute*), 32
dismiss_on_outside_click()
    built-in function, 49
dismiss_on_scroll()
    built-in function, 49
dynamic_vars, 56

## E

Easing, 27
Effect (*built-in class*), 21
element (*ZodList attribute*), 89
end(), 34
error (*AsyncCall property*), 42
error() (*Logger method*), 33
error_form (*routing attribute*), 56
extend() (*ZodTypedDict method*), 83

## F

finish(), 26
format (*Logger attribute*), 32

format_selected_text()
    built-in function, 9

## G

get() (*LocalStorageWrapper method*), 74
get_bounding_rect()
    built-in function, 25
get_format_params() (*Logger method*), 33
getKeyframes(), 26
getTiming(), 26

## H

has_popover()
    built-in function, 49
height_in() (*Transition class method*), 26
height_out() (*Transition class method*), 26

## I

IndexedDBWrapper (*built-in class*), 73
info() (*Logger method*), 33
is_animating()
    built-in function, 25
items() (*LocalStorageWrapper method*), 74

## K

keyof() (*ZodTypedDict method*), 83
keys() (*LocalStorageWrapper method*), 74

## L

level (*Logger attribute*), 32
LocalStorageWrapper (*built-in class*), 73
log() (*Logger method*), 33
Logger (*built-in class*), 32
logger (*routing attribute*), 58

## M

merge() (*ZodTypedDict method*), 83

## N

name (*Logger attribute*), 32
nonempty() (*ZodList method*), 89

## O

omit() (*ZodTypedDict method*), 84
on_error() (*AsyncCall method*), 42
on_form_load(), 55
on_navigation() (*routing method*), 55
on_result() (*AsyncCall method*), 42
oncancel, 27
onfinish, 27
onremove, 27

## P

parse(), 92
partial() (*ZodTypedDict method*), 85
passthrough() (*ZodTypedDict method*), 86
pause(), 26
persist(), 26
pick() (*ZodTypedDict method*), 84
play(), 26
playbackRate, 27
pop(), 49
pop() (*LocalStorageWrapper method*), 74
popover(), 48

## R

repeat()
    built-in function, 42
required() (*ZodTypedDict method*), 85
result (*AsyncCall property*), 42
reverse(), 27
routing.add_to_cache()
    built-in function, 58
routing.clear_cache()
    built-in function, 58
routing.get_cache()
    built-in function, 58
routing.get_url_components()
    built-in function, 57
routing.get_url_dict()
    built-in function, 57
routing.get_url_hash()
    built-in function, 57
routing.get_url_pattern()
    built-in function, 57
routing.go()
    built-in function, 58
routing.go_back()
    built-in function, 58
routing.launch()
    built-in function, 57
routing.load_error_form()
    built-in function, 57
routing.NavigationExit, 57
routing.on_session_expired()
    built-in function, 58
routing.redirect()
    built-in function, 56
routing.reload_page()
    built-in function, 58
routing.remove_from_cache()
    built-in function, 57
routing.route()
    built-in function, 56
routing.set_url_hash()
    built-in function, 57

routing.set_warning_before_app_unload()
    built-in function, 58
routing.template()
    built-in function, 55

## S

safe_parse(), 92
set_default_container()
    built-in function, 49
set_default_max_width()
    built-in function, 49
shape (*ZodTypedDict attribute*), 83
start(), 34
status (*AsyncCall property*), 42
StorageWrapper (*built-in class*), 73
store() (*LocalStorageWrapper method*), 74
stream (*Logger attribute*), 32
strict() (*ZodTypedDict method*), 87
strip() (*ZodTypedDict method*), 87

## T

Transition (*built-in class*), 21

## U

update() (*LocalStorageWrapper method*), 74
updatePlaybackRate(), 27
url_dict, 56
url_hash (*routing attribute*), 56
url_pattern, 56

## V

values() (*LocalStorageWrapper method*), 74

## W

wait(), 27
wait_for()
    built-in function, 25, 42
warning() (*Logger method*), 33
width_in() (*Transition class method*), 26
width_out() (*Transition class method*), 26

## Z

ZodList (*built-in class*), 89
ZodTypedDict (*built-in class*), 83