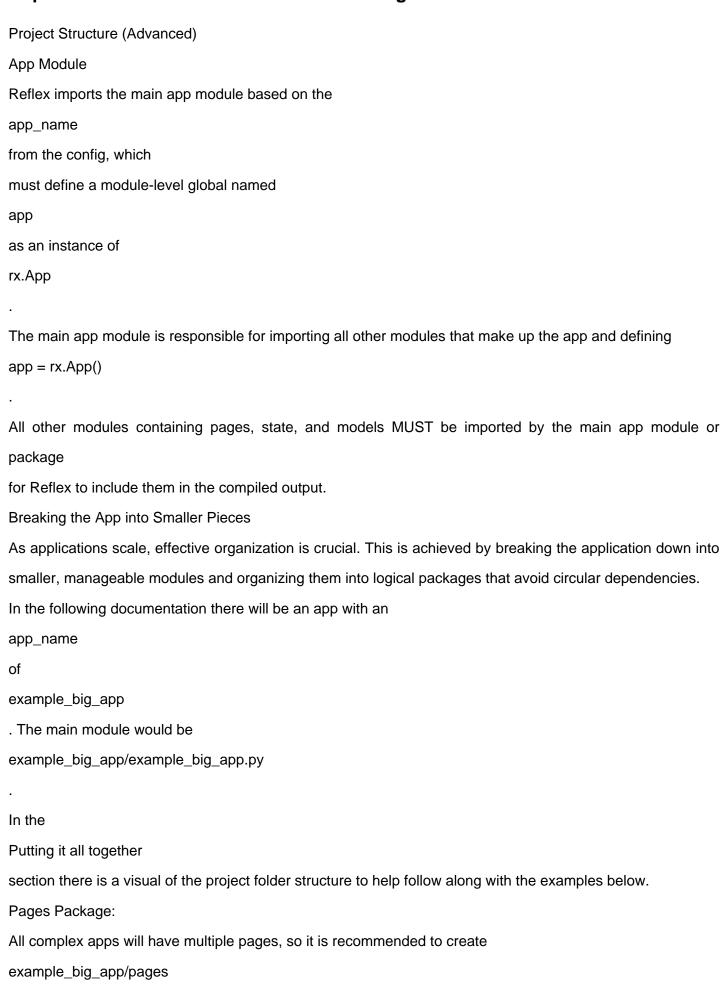
https://reflex.dev/docs/advanced-onboarding/code-structure



as a package.

This package should contain one module per page in the app.

If a particular page depends on the state, the substate should be defined in the same module as the page.

The page-returning function should be decorated with

rx.page()

to have it added as a route in the app.

Templating:

Most applications maintain a consistent layout and structure across pages. Defining this common structure in a separate module facilitates easy sharing and reuse when constructing individual pages.

Best Practices

Factor out common frontend UI elements into a function that returns a component.

If a function accepts a function that returns a component, it can be used as a decorator as seen below.

The

@template

decorator should appear below the

@rx.page

decorator and above the page-returning function. See the

Posts Page

code for an example.

State Management

Most pages will use State in some capacity. You should avoid adding vars to a

shared state that will only be used in a single page. Instead, define a new

subclass of

rx.State

and keep it in the same module as the page.

Accessing other States

As of Reflex 0.4.3, any event handler can get access to an instance of any other

substate via the

get_state

API. From a practical perspective, this means that

state can be split up into smaller pieces without requiring a complex

inheritance hierarchy to share access to other states.

In previous releases, if an app wanted to store settings in

SettingsState

with

a page or component for modifying them, any other state with an event handler that needed to access those settings would have to inherit from SettingsState

even if the other state was mostly orthogonal. The other state would also now always have to load the settings, even for event handlers that didn't need to access them.

A better strategy is to load the desired state on demand from only the event handler which needs access to the substate.

A Settings Component:

A Post Page:

This page loads the

SettingsState

to determine how many posts to display per page

and how often to refresh.

Common State:

Common

states and substates that are shared by multiple pages or components should be implemented in a separate module to avoid circular imports. This module should not import other modules in the app.

Component Reusability

The primary mechanism for reusing components in Reflex is to define a function that returns the component, then simply call it where that functionality is needed.

Component functions typically should not take any State classes as arguments, but prefer to import the needed state and access the vars on the class directly.

Memoize Functions for Improved Performance

In a large app, if a component has many subcomponents or is used in a large number of places, it can improve compile and runtime performance to memoize the function with the

@lru_cache

decorator.

To memoize the

foo

component to avoid re-creating it many times simply add

@Iru cache

to the function definition, and the component will only be created once per unique set of arguments.

example_big_app/components

This package contains reusable parts of the app, for example headers, footers,

and menus. If a particular component requires state, the substate may be defined

in the same module for locality. Any substate defined in a component module

should only contain fields and event handlers pertaining to that individual

component.

External Components

Reflex 0.4.3 introduced support for the

reflex component

CLI commands

, which makes it easy

to bundle up common functionality to publish on PyPI as a standalone Python package

that can be installed and used in any Reflex app.

When wrapping npm components or other self-contained bits of functionality, it can be helpful

to move this complexity outside the app itself for easier maintenance and reuse in other apps.

Database Models:

It is recommended to implement all database models in a single file to make it easier to define relationships

and understand the entire schema.

However, if the schema is very large, it might make sense to have a

models

package with individual models defined in their own modules.

At any rate, defining the models separately allows any page or component to import and use them without

circular imports.

Top-level Package:

This is a great place to import all state, models, and pages that should be part of the app.

Typically, components and helpers do not need to imported, because they will be imported by

pages that use them (or they would be unused).

If any pages are not imported here, they will not be compiled as part of the app.

example_big_app/example_big_app.py

This is the main app module. Since everything else is defined in other modules, this file becomes very simple.

File Management

There are two categories of non-code assets (media, fonts, stylesheets,

```
documents) typically used in a Reflex app.
assets
The
assets
directory is used for
static
files that should be accessible
relative to the root of the frontend (default port 3000). When an app is deployed in
production mode, changes to the assets directory will NOT be available at runtime!
When referencing an asset, always use a leading forward slash, so the
asset can be resolved regardless of the page route where it may appear.
uploaded_files
If an app needs to make files available dynamically at runtime, it is
recommended to set the target directory via
REFLEX_UPLOADED_FILES_DIR
environment variable (default
./uploaded_files
), write files relative to the
path returned by
rx.get_upload_dir()
, and create working links via
rx.get_upload_url(relative_path)
Uploaded files are served from the backend (default port 8000) via
/_upload/<relative_path>
Putting it all together
Based on the previous discussion, the recommended project layout look like this.
Key Takeaways
Like any other Python project,
split up the app into modules and packages
to keep the codebase organized and manageable.
Using smaller modules and packages makes it easier to
reuse components and state
across the app
```

without introducing circular dependencies.

Create

individual functions

to encapsulate units of functionality and

reuse them

where needed.

https://reflex.dev/docs/advanced-onboarding/configuration

Configuration
Reflex apps can be configured using a configuration file, environment variables, and command line
arguments.
Configuration File
Running
reflex init
will create an
rxconfig.py
file in your root directory.
You can pass keyword arguments to the
Config
class to configure your app.
For example:
See the
config reference
for all the parameters available.
Environment Variables
You can override the configuration file by setting environment variables.
For example, to override the
frontend_port
setting, you can set the
FRONTEND_PORT
environment variable.
Command Line Arguments
Finally, you can override the configuration file and environment variables by passing command line
arguments to
reflex run
•
See the
CLI reference

for all the arguments available.

Customizable App Data Directory

REFLEX_DIR
environment variable can be set, which allows users to set the location where Reflex writes
Bun and NodeJS.
By default we use Platform specific directories:
On windows,
C:/Users/ <username>/AppData/Local/reflex</username>
is used.
On macOS,
~/Library/Application Support/reflex
is used.
On linux,
~/.local/share/reflex
is used.

helper tools like

The

https://reflex.dev/docs/advanced-onboarding/how-reflex-works

How Reflex Works

We'll use the following basic app that displays Github profile images as an example to explain the different parts of the architecture.

The Reflex Architecture

Full-stack web apps are made up of a frontend and a backend. The frontend is the user interface, and is served as a web page that runs on the user's browser. The backend handles the logic and state management (such as databases and APIs), and is run on a server.

In traditional web development, these are usually two separate apps, and are often written in different frameworks or languages. For example, you may combine a Flask backend with a React frontend. With this approach, you have to maintain two separate apps and end up writing a lot of boilerplate code to connect the frontend and backend.

We wanted to simplify this process in Reflex by defining both the frontend and backend in a single codebase, while using Python for everything. Developers should only worry about their app's logic and not about the low-level implementation details.

TLDR

Under the hood, Reflex apps compile down to a

React

frontend app and a

FastAPI

backend app. Only the UI is compiled to Javascript; all the app logic and state management stays in Python and is run on the server. Reflex uses

WebSockets

to send events from the frontend to the backend, and to send state updates from the backend to the frontend. The diagram below provides a detailed overview of how a Reflex app works. We'll go through each part in more detail in the following sections.

Frontend

We wanted Reflex apps to look and feel like a traditional web app to the end user, while still being easy to build and maintain for the developer. To do this, we built on top of mature and popular web technologies.

When you

reflex run

your app, Reflex compiles the frontend down to a single-page

Next.js

app and serves it on a port (by default

3000

) that you can access in your browser.

The frontend's job is to reflect the app's state, and send events to the backend when the user interacts with the UI. No actual logic is run on the frontend.

Components

Reflex frontends are built using components that can be composed together to create complex UIs. Instead of using a templating language that mixes HTML and Python, we just use Python functions to define the UI.

In our example app, we have components such as

rx.hstack

rx.avatar

, and

rx.input

. These components can have different

props

that affect their appearance and functionality - for example the

rx.input

component has a

placeholder

prop to display the default text.

We can make our components respond to user interactions with events such as

on_blur

, which we will discuss more below.

Under the hood, these components compile down to React components. For example, the above code compiles down to the following React code:

Many of our core components are based on

Radix

, a popular React component library. We also have many other components for graphing, datatables, and more.

We chose React because it is a popular library with a huge ecosystem. Our goal isn't to recreate the web ecosystem, but to make it accessible to Python developers.

This also lets our users bring their own components if we don't have a component they need. Users can wrap their own React components

and then publish them for others to use. Over time we will build out our third party component ecosystem so that users can easily find and use components that others have built. Styling We wanted to make sure Reflex apps look good out of the box, while still giving developers full control over the appearance of their app. We have a core theming system that lets you set high level styling options such as dark mode and accent color throughout your app to give it a unified look and feel. Beyond this, Reflex components can be styled using the full power of CSS. We leverage the **Emotion** library to allow "CSS-in-Python" styling, so you can pass any CSS prop as a keyword argument to a component. This includes responsive props by passing a list of values. **Backend** Now let's look at how we added interactivity to our apps. In Reflex only the frontend compiles to Javascript and runs on the user's browser, while all the state and logic stays in Python and is run on the server. When you reflex run , we start a FastAPI server (by default on port 8000) that the frontend connects to through a websocket. All the state and logic are defined within a State class. The state is made up of vars and event handlers

Vars are any values in your app that can change over time. They are defined as class attributes on your State class, and may be any Python type that can be serialized to JSON. In our example,

url

and

profile_image

are vars.

Event handlers are methods in your

State

class that are called when the user interacts with the UI. They are the only way that we can modify the vars in Reflex, and can be called in response to user actions, such as clicking a button or typing in a text box. In our example,

set_profile

is an event handler that updates the

url

and

profile_image

vars.

Since event handlers are run on the backend, you can use any Python library within them. In our example, we use the

requests

library to make an API call to Github to get the user's profile image.

Event Processing

Now we get into the interesting part - how we handle events and state updates.

Normally when writing web apps, you have to write a lot of boilerplate code to connect the frontend and backend. With Reflex, you don't have to worry about that - we handle the communication between the frontend and backend for you. Developers just have to write their event handler logic, and when the vars are updated the UI is automatically updated.

You can refer to the diagram above for a visual representation of the process. Let's walk through it with our Github profile image example.

Event Triggers

The user can interact with the UI in many ways, such as clicking a button, typing in a text box, or hovering over an element. In Reflex, we call these

event triggers

In our example we bind the

event trigger to the

set_profile

on blur

event handler. This means that when the user types in the input field and then clicks away, the

set_profile

event handler is called.

Event Queue

On the frontend, we maintain an event queue of all pending events. An event consists of three major pieces of data:

client token

: Each client (browser tab) has a unique token to identify it. This let's the backend know which state to update.

event handler

: The event handler to run on the state.

arguments

: The arguments to pass to the event handler.

Let's assume I type my username "picklelo" into the input. In this example, our event would look something like this:

On the frontend, we maintain an event queue of all pending events.

When an event is triggered, it is added to the queue. We have a

processing

flag to make sure only one event is processed at a time. This ensures that the state is always consistent and there aren't any race conditions with two event handlers modifying the state at the same time.

There are exceptions to this, such as

background events

which allow you to run events in the background without blocking the UI.

Once the event is ready to be processed, it is sent to the backend through a WebSocket connection.

State Manager

Once the event is received, it is processed on the backend.

Reflex uses a

state manager

which maintains a mapping between client tokens and their state. By default, the state manager is just an

in-memory dictionary, but it can be extended to use a database or cache. In production we use Redis as our state manager.

Event Handling

Once we have the user's state, the next step is to run the event handler with the arguments.

In our example, the

set_profile

event handler is run on the user's state. This makes an API call to Github to get the user's profile image, and then updates the state's

url

and

profile_image

vars.

State Updates

Every time an event handler returns (or

yields

), we save the state in the state manager and send the

state updates

to the frontend to update the UI.

To maintain performance as your state grows, internally Reflex keeps track of vars that were updated during the event handler (

dirty vars

). When the event handler is done processing, we find all the dirty vars and create a state update to send to the frontend.

In our case, the state update may look something like this:

We store the new state in our state manager, and then send the state update to the frontend. The frontend then updates the UI to reflect the new state. In our example, the new Github profile image is displayed.

https://reflex.dev/docs/ai-builder/features/deploy-app

Deploy your App It is easy to deploy your app into production from Reflex Build to Reflex Cloud. Simply click the Deploy button in the bottom right corner of Reflex Build, as shown below: When deploying you can set the following options: App Name : The name of your app Hostname : Set your url by setting your hostname, i.e. if you set myapp as your hostname, your app will be available at myapp.reflex.run Region : The regions where your app will be deployed VM Size : The size of the VM where your app will be deployed Secrets : The environment variables that will be set for your app, you can load the variables currently being used by your app by clicking the

Load from settings

button

https://reflex.dev/docs/ai-builder/features/download-app

Download your App

It is easy to download your app to work on locally or self-host. (It is recommended to use the GitHub integration, but if this is not possible, you can download your app to work on locally.)

Simply click the

Download

button in the bottom right corner of Reflex Build, as shown below:

https://reflex.dev/docs/ai-builder/features/environment-variables

Environment Variables (Secrets)

It is possible to add environment variables to your app. This is useful for storing secrets such as API keys, and other sensitive information.

Adding Environment Variables

You can add environment variables to your app by clicking the

Secrets

button at the bottom of the chat input box, as seen below:

After you add the environment variables the AI now has context of these and you can prompt it to use them in your code.

You can also add environment variables after your app is built, by again clicking the

Secrets

button at the bottom of the chat input box on the generation page.

https://reflex.dev/docs/ai-builder/features/ide

Reflex Build's IDE

Reflex Build includes a powerful, in-browser IDE designed to streamline the entire development processâ€"from writing code

to deploying your app. With an intuitive layout, real-time editing, and seamless integration with the rest of the platform, the IDE empowers users to stay focused and productive without ever leaving the browser.

IDE Features

Real-Time Editing

Changes you make in the editor are immediately reflected across your projectâ€"no manual saves or rebuilds required. Stay in flow and iterate faster.

File & Folder Management

Easily create, rename, or delete files and folders directly in the workspace. The file tree gives you full visibility into your application structure at all times.

Drag-and-Drop File Upload

Seamlessly import files into your project by dragging them straight into the editor. Whether you're adding assets, scripts, or config files, it's fast and intuitiveâ€"no extra clicks required.

Context-Aware Code Editor

The built-in code editor supports syntax highlighting, inline error detection, and Al-assisted suggestions to help you write clean, efficient code with confidence.

One-Click Deployment

From the IDE, you can deploy your app with a single click. No terminal, no external toolsâ€"just build and ship straight from your browser.

https://reflex.dev/docs/ai-builder/features/image-as-prompt

Use Images as a prompt

Uploading an image (screenshot) of a website (web) app of what you are looking to build gives the AI really good context.

This is the recommended way to start an app generation.

Below is a GIF showing how to upload an image to the AI Builder:

The advised prompt to use is:

Build an app from a reference image

https://reflex.dev/docs/ai-builder/features/installing-external-packages

Installing External Packages

Reflex Build allows you to install external python packages to use in your app. This is useful if you want to
use a package that is not included in the default Reflex Build environment. Examples might include
openai
,
langsmith

requests

, etc.

There are two ways to install external packages:

Through the Chat Interface

: You can ask the AI to install a package for you.

Add to the

requirements.txt

file

: You can add the package to the

requirements.txt

file and then save the app. This will install the package in your app's environment.

Installing through the Chat Interface

Enter the name of the package you want to install in the chat interface. The Al will then install the package for you.

Installing through the requirements.txt file

Add the package to the

requirements.txt

file and then save the app. This will install the package in your app's environment and recompile your app.

https://reflex.dev/docs/ai-builder/features/templates

Templates

Reflex has many certified templates, seen on the

Trending

tab of the Reflex Build, that can be used to kickstart your app. You can also use any app created by the community as a template.

Using a Template

To use a template, simply click the template and then in the bottom right corner of the app click the

Fork

button. This will create a copy of the template in your own account. You can then edit the app as you like with further prompting.

Below is an example of how to use a template:

Templates are great to get started if they have similar UI to what you are looking to build. You can then add your own data to the app.

https://reflex.dev/docs/ai-builder/integrations/database

Connecting to a Database

Connecting to a database is critical to give your app access to real data. This section will cover how to connect to a database using the Al Builder.

To connect to a database you will need a

DB_URI

. Reflex.build currently supports

postgresql

and

mysql

databases.

This is what it looks like for a Postgres database:

DB URI (More Details)

You can also use a MySQL database. The connection string looks like this:

Connecting your Database before the app is generated

You can add your

Database URI

at the start of your generation as shown below.

Here if you wanted to build a dashboard for example we recommend a prompt as follows:

Build a dashboard around my database data

Connecting your Database after the app is generated

You can add your

Database URI

after you've already generated an app or directly from a template that you have forked as shown below.

Here if you wanted to hook up your data correctly we recommend a prompt as follows:

Use the database I added to rewrite the dashboard to display my expense reporting data, keep the existing layout charts and structure the same

https://reflex.dev/docs/ai-builder/integrations/github

Connecting to Github

The Github integration is important to make sure that you don't lose your progress. It also allows you to revert to previous versions of your app.

The GitHub integration allows you to:

Save your app progress

Work on your code locally and push your local changes back to Reflex.Build

Github Commit History

The commit history is a great way to see the changes that you have made to your app. You can also revert to previous versions of your app from here.

https://reflex.dev/docs/ai-builder/overview

Overview

https://reflex.dev/docs/ai-builder/overview/what-is-reflex-build

What Is Reflex Build

Reflex Build is an Al-powered platform that empowers users of all skill levels to create full-stack web applications

without writing any codeâ€"just by describing their ideas in plain English. Instead of hiring developers, users can

instantly generate web apps or websites, turning ideas into functional apps as quickly as possible.

Reflex Build provides everything you need to create stunning websites, front-end interfaces, and full-stack web

applicationsâ€"all from a single browser tab, with no installation required. It includes AI-powered coding tools, real-time collaboration (currently in beta), and easy project sharing to give you a head start on your app development

journey.

Feature Overview

Reflex Build provides a streamlined interface for building AI applications. The

Project Menu Bar

helps you manage sessions and stored variables, while the

Chat Area

displays real-time prompts, edits, and file generations. The

Application Workspace

organizes your project structure, and the

Code Editor

allows direct, instant code editing. Key actions like deploy and share are accessible via the

Bottom Menu Bar

, and the

Preview Tab

lets you view and interact with your live app at any time.

Project Menu Bar

Browse previously built applications, create new sessions, store database variables, and much more!

Chat Area

See your prompts in action with visual cues, editing notifications, and file generations every step of the way.

Application Workspace

Your workspace contains all the folders and files of your application. You can add new files and folders as

well!

Code Editor

The code editor displays the current selected file. You can edit the code directly and save it instantly.

Bottom Menu Bar

This menu contains important actions such as deploying, downloading, and sharing your application.

Preview Tab

The preview tab showcases a live application. You can navigate to other applications directly from this tab, refresh the app, and even view it in full screen.

Interface Highlights

Reflex Build's interface is designed for clarity and efficiency. The

Project Menu Bar

helps you manage sessions, apps, and variables. The

Chat Area

shows prompts in action with visual feedback and file generation. In the

Application Workspace

, you can view and organize your project files. The

Code Editor

allows quick, direct edits with instant saving. Use the

Bottom Menu Bar

for key actions like deploy and download. The

Preview Tab

lets you interact with a live version of your app, including refresh and full-screen options.

Database Integration

Automatically integrate your database

into your application with ease

Secure Secrets

Safely manage your API keys and tokens

with a built in secrets manager

Live Preview

See all application changes in real-time

with our interactive preview tab

Quick Download

Download your complete project files

with just a single click operation

Easy Deployment

Deploy your application to production

with just a single click process

Manual File Editing

Edit your project files directly

with our intuitive code editor

Al Package Manager

Let AI handle your package installations

via natural prompting

Smart Prompting

Get better development results

with Al-optimized prompt templates

https://reflex.dev/docs/ai-builder/prompting/breaking-up-complex-prompts

Breaking up complex prompts
Incremental Prompting
Asking for incremental, smaller changes leads to better results, rather than asking for everything all in a
single huge prompt. It's better to take it step-by-step rather than give the AI complex tasks all at once.
Example 1
Too Complex:
Create a data visualization dashboard that includes user authentication, data integration, multiple charts,
real-time updates, and export options.
Better Approach:
Prompt 1:
Design a simple dashboard layout.
Prompt 2:
Now let's add a bar chart for visualizing sales data.
Prompt 3:
Now add user authentication.
Prompt 4:
Now integrate data from an external API.
Prompt 5:
Now add real-time updates for the chart.
Prompt 6:
Now add export options for the dashboard data.
Example 2
Too Complex:
Create an app that takes in data, processes it, generates reports, sends notifications, and allows users to
filter results by various criteria.
Better Approach:
Prompt 1:
Create an app that takes in data
Prompt 2:
Now add logic to process the data.
Prompt 3:
Now add a feature to generate reports.

Prompt 4:

Now add a feature to send notifications.

Prompt 5:

Now add a feature to filter results by various criteria.

https://reflex.dev/docs/api-reference/app

App

reflex.app.App

The main Reflex app that encapsulates the backend and frontend.

Every Reflex app needs an app defined in its main module.

Methods

Signature

Description

frontend_exception_handler(exception: 'Exception') -> 'None'

Default frontend exception handler function.

backend_exception_handler(exception: 'Exception') -> 'EventSpec'

Default backend exception handler function.

add_page(self, component: 'Component | ComponentCallable | None' = None, route: 'str | None' = None, title: 'str | Var | None' = None, description: 'str | Var | None' = None, image: 'str' = 'favicon.ico', on_load:

'EventType[()] | None' = None, meta: 'list[dict[str, str]]' = [], context: 'dict[str, Any] | None' = None)

Add a page to the app.

If the component is a callable, by default the route is the name of the

function. Otherwise, a route must be provided.

get_load_events(self, path: 'str') -> 'list[IndividualEventType[()]]'

Get the load events for a route.

add_all_routes_endpoint(self)

Add an endpoint to the app that returns all the routes.

modify_state(self, token: 'str') -> 'AsyncIterator[BaseState]'

Modify the state out of band.

https://reflex.dev/docs/api-reference/base

Base

reflex.base.Base

The base class subclassed by all Reflex classes.

This class wraps Pydantic and provides common methods such as serialization and setting fields.

Any data structure that needs to be transferred between the

frontend and backend should subclass this class.

Methods

Signature

Description

json(self) -> str

Convert the object to a json string.

set(self, **kwargs: object)

Set multiple fields and return the object.

https://reflex.dev/docs/api-reference/browser-javascript

Browser Javascript

Reflex compiles your frontend code, defined as python functions, into a Javascript web application that runs in the user's browser. There are instances where you may need to supply custom javascript code to interop with Web APIs, use certain third-party libraries, or wrap low-level functionality that is not exposed via Reflex's Python API.

Avoid Custom Javascript

Executing Script

There are four ways to execute custom Javascript code into your Reflex app:

rx.script

- Injects the script via

next/script

for efficient loading of inline and external Javascript code. Described further in the component library

These components can be directly included in the body of a page, or they may

be passed to

rx.App(head_components=[rx.script(...)])

to be included in

the

<Head>

tag of all pages.

rx.call_script

- An event handler that evaluates arbitrary Javascript code,

and optionally returns the result to another event handler.

These previous two methods can work in tandem to load external scripts and then

call functions defined within them in response to user events.

The following two methods are geared towards wrapping components and are

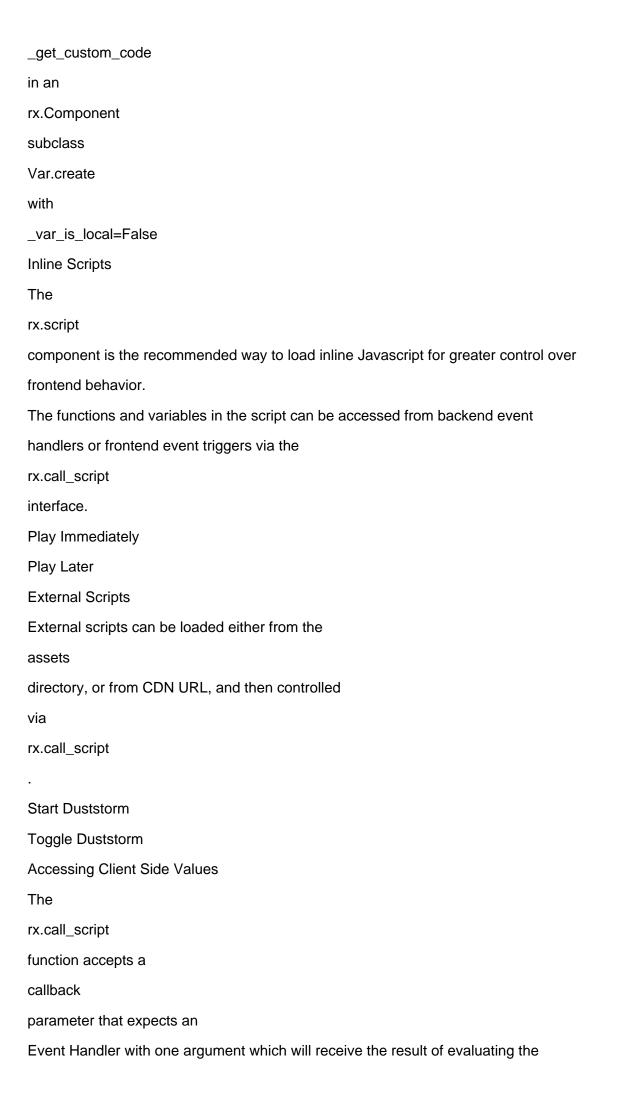
described with examples in the

Wrapping React

section.

_get_hooks

and



Javascript code. This can be used to access client-side values such as the window.location or current scroll location, or any previously defined value. **Update Values** Scroll Position: {} window.location: {} Allowed Callback Values **Using React Hooks** To use React Hooks directly in a Reflex app, you must subclass rx.Component typically rx.Fragment is used when the hook functionality has no visual element. The hook code is returned by the add_hooks method, which is expected to return a list[str] containing Javascript code which will be inserted into the page component (i.e the render function itself). For supporting code that must be defined outside of the component render function, use _get_custom_code The following example uses useEffect to register global hotkeys on the document object, and then triggers an event when a specific key is pressed. Press a, s, d or w to trigger an event Last watched key pressed: This snippet can also be imported through pip:

reflex-global-hotkey

.

https://reflex.dev/docs/api-reference/browser-storage

Browser Storage rx.Cookie Represents a state Var that is stored as a cookie in the browser. Currently only supports string values. **Parameters** name : The name of the cookie on the client side. path : The cookie path. Use to make the cookie accessible on all pages. max_age : Relative max age of the cookie in seconds from when the client receives it. domain : Domain for the cookie (e.g., sub.domain.com or .allsubdomains.com). secure : If the cookie is only accessible through HTTPS. same_site : Whether the cookie is sent with third-party requests. Can be one of (True False None lax strict

).

The default value of a Cookie is never set in the browser! **Accessing Cookies** Cookies are accessed like any other Var in the state. If another state needs access to the value of a cookie, the state should be a substate of the state that defines the cookie. Alternatively the get_state API can be used to access the other state. For rendering cookies in the frontend, import the state that defines the cookie and reference it directly. Two separate states should avoid defining rx.Cookie with the same name. rx.remove cookies Remove a cookie from the client's browser. Parameters: key : The name of cookie to remove. This event can also be returned from an event handler: rx.LocalStorage Represents a state Var that is stored in localStorage in the browser. Currently only supports string values. **Parameters** name : The name of the storage key on the client side. sync : Boolean indicates if the state should be kept in sync across tabs of the same browser. Syncing Vars Because LocalStorage applies to the entire browser, all LocalStorage Vars are automatically shared across tabs. The sync parameter controls whether an update in one tab should be actively propagated to other tabs without requiring a navigation or page refresh event.

rx.remove_local_storage

Remove a local storage item from the client's browser.

Parameters

key

: The key to remove from local storage.

This event can also be returned from an event handler:

rx.clear local storage()

Clear all local storage items from the client's browser. This may affect other apps running in the same domain or libraries within your app that use local storage.

rx.SessionStorage

Represents a state Var that is stored in sessionStorage in the browser. Similar to localStorage, but the data is cleared when the page session ends (when the browser/tab is closed). Currently only supports string values.

Parameters

name

: The name of the storage key on the client side.

Session Persistence

SessionStorage data is cleared when the page session ends. A page session lasts as long as the browser is open and survives page refreshes and restores, but is cleared when the tab or browser is closed.

Unlike LocalStorage, SessionStorage is isolated to the tab/window in which it was created, so it's not shared with other tabs/windows of the same origin.

rx.remove_session_storage

Remove a session storage item from the client's browser.

Parameters

key

: The key to remove from session storage.

This event can also be returned from an event handler:

rx.clear session storage()

Clear all session storage items from the client's browser. This may affect other apps running in the same domain or libraries within your app that use session storage.

Serialization Strategies

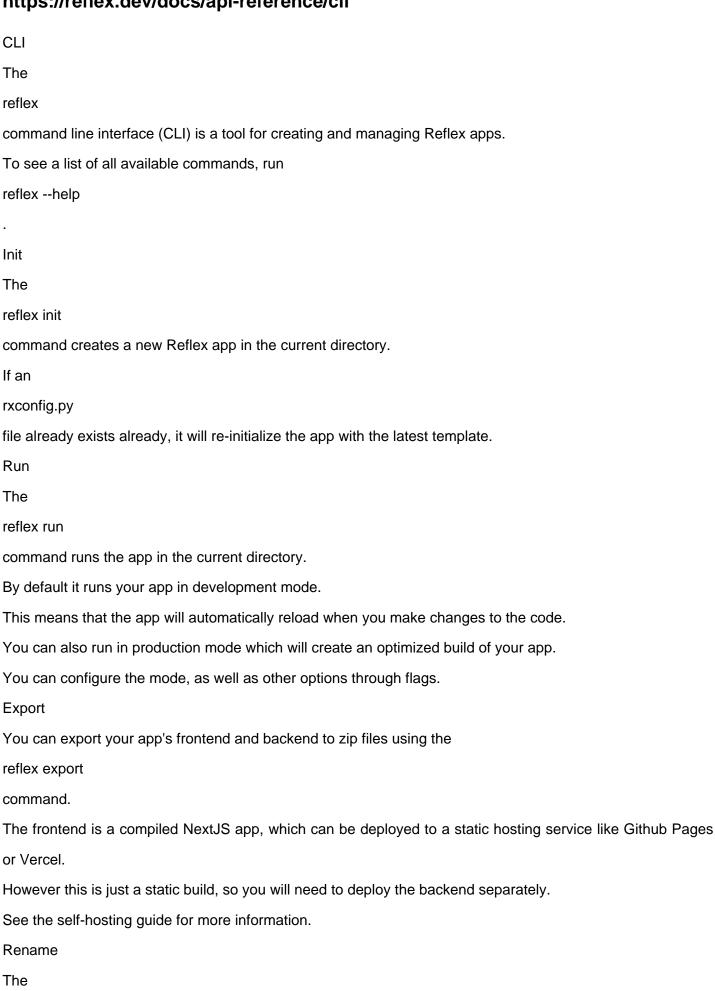
If a non-trivial data structure should be stored in a

Cookie

LocalStorage
, or
SessionStorage
var it needs to be serialized before and after storing it. It is recommended to use a pydantic class for the data
which provides simple serialization helpers and works recursively in complex object structures.
App Settings
Comparison of Storage Types
Here's a comparison of the different client-side storage options in Reflex:
Feature
rx.Cookie
rx.LocalStorage
rx.SessionStorage
Persistence
Until cookie expires
Until explicitly deleted
Until browser/tab is closed
Storage Limit
~4KB
~5MB
~5MB
Sent with Requests
Yes
No
No
Accessibility
Server & Client
Client Only
Client Only
Expiration
Configurable
Never
End of session
Scope

Configurable (domain, path)
Origin (domain)
Tab/Window
Syncing Across Tabs
No
Yes (with sync=True)
No
Use Case
Authentication, Server-side state
User preferences, App state
Temporary session data
When to Use Each Storage Type
Use rx.Cookie When:
You need the data to be accessible on the server side (cookies are sent with HTTP requests)
You're handling user authentication
You need fine-grained control over expiration and scope
You need to limit the data to specific paths in your app
Use rx.LocalStorage When:
You need to store larger amounts of data (up to ~5MB)
You want the data to persist indefinitely (until explicitly deleted)
You need to share data between different tabs/windows of your app
You want to store user preferences that should be remembered across browser sessions
Use rx.SessionStorage When:
You need temporary data that should be cleared when the browser/tab is closed
You want to isolate data to a specific tab/window
You're storing sensitive information that shouldn't persist after the session ends
You're implementing per-session features like form data, shopping carts, or multi-step processes
You want to persist data for a state after Redis expiration (for server-side state that needs to survive longer
than Redis TTL)

https://reflex.dev/docs/api-reference/cli



reflex rename
command allows you to rename your Reflex app. This updates the app name in the configuration files.
Cloud
The
reflex cloud
command provides access to the Reflex Cloud hosting service. It includes subcommands for managing apps,
projects, secrets, and more.
For detailed documentation on Reflex Cloud and deployment, see the
Cloud Quick Start Guide
Script
The
reflex script
command provides access to helper scripts for Reflex development.

https://reflex.dev/docs/api-reference/component

Component

reflex.components.component.Component

A component with style, event trigger and other props.

Methods

Signature

Description

add_imports(self) -> 'ImportDict | list[ImportDict]'

Add imports for the component.

This method should be implemented by subclasses to add new imports for the component.

Implementations do NOT need to call super(). The result of calling add_imports in each parent class will be merged internally.

add_hooks(self) -> 'list[str | Var]'

Add hooks inside the component function.

Hooks are pieces of literal Javascript code that is inserted inside the React component function.

Each logical hook should be a separate string in the list.

Common strings will be deduplicated and inserted into the component function only once, so define const variables and other identical code in their own strings to avoid defining the same const or hook multiple times.

If a hook depends on specific data from the component instance, be sure to use unique values inside the string to _avoid_ deduplication.

Implementations do NOT need to call super(). The result of calling add_hooks in each parent class will be merged and deduplicated internally. add_custom_code(self) -> 'list[str]'

Add custom Javascript code into the page that contains this component.

Custom code is inserted at module level, after any imports.

Each string of custom code is deduplicated per-page, so take care to avoid defining the same const or function differently from different component instances.

Custom code is useful for defining global functions or constants which can then be referenced inside hooks or used by component vars.

Implementations do NOT need to call super(). The result of calling add_custom_code in each parent class will be merged and deduplicated internally.

get_event_triggers(cls) -> 'dict[str, types.ArgsSpec | Sequence[types.ArgsSpec]]'

Get the event triggers for the component.

get_props(cls) -> 'set[str]'

Get the unique fields for the component.

get_initial_props(cls) -> 'set[str]'

Get the initial props to set for the component.

create(cls: 'type[T]', *children, **props) -> 'T'

Create the component.

add_style(self) -> 'dict[str, Any] | None'

Add style to the component.

Downstream components can override this method to return a style dict that will be applied to the component.

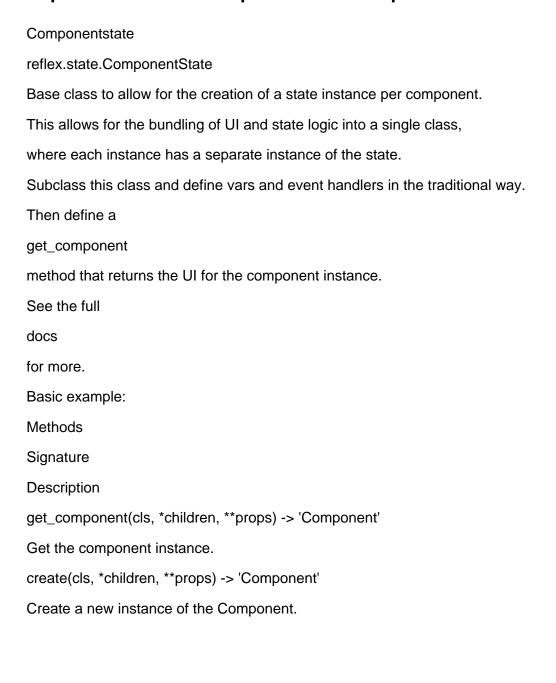
render(self) -> 'dict'

Render the component.

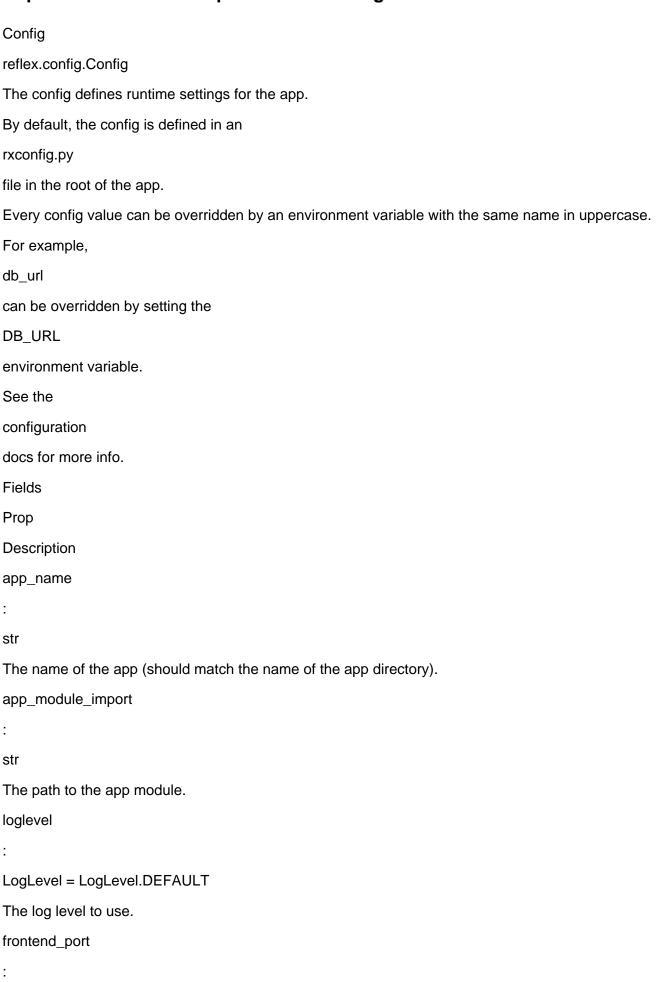
get_ref(self) -> 'str | None'

Get the name of the ref for the component.

https://reflex.dev/docs/api-reference/componentstate



https://reflex.dev/docs/api-reference/config



int The port to run the frontend on. NOTE: When running in dev mode, the next available port will be used if this is taken. frontend_path : str The path to run the frontend on. For example, "/app" will run the frontend on http://localhost:3000/app backend_port int The port to run the backend on. NOTE: When running in dev mode, the next available port will be used if this is taken. api_url str = http://localhost:8000 The backend url the frontend will connect to. This must be updated if the backend is hosted elsewhere, or in production. deploy_url str = http://localhost:3000 The url the frontend will be hosted on. backend_host str = 0.0.0.0The url the backend will be hosted on. db_url str = sqlite:///reflex.db The database url used by rx.Model.

The async database url used by rx.Model.

async_db_url

:

str

```
redis_url
str
The redis url
telemetry_enabled
bool = True
Telemetry opt-in.
bun_path
Path = /home/runner/.local/share/reflex/bun/bin/bun
The bun path
static_page_generation_timeout
int = 60
Timeout to do a production build of a frontend page.
cors_allowed_origins
str = ['*']
List of origins that are allowed to connect to the backend API.
react_strict_mode
bool = True
Whether to use React strict mode.
frontend_packages
str
Additional frontend packages to install.
state_manager_mode
StateManagerMode = StateManagerMode.DISK
Indicate which type of state manager to use
redis_lock_expiration
```

```
int = 10000
Maximum expiration lock time for redis state manager
redis_lock_warning_threshold
int = 1000
Maximum lock time before warning for redis state manager.
redis_token_expiration
int = 3600
Token expiration time for redis state manager
env_file
str
Path to file containing key-values pairs to override in the environment; Dotenv format.
state_auto_setters
bool = True
Whether to automatically create setters for state base vars
show_built_with_reflex
bool
Whether to display the sticky "Built with Reflex" badge on all pages.
is_reflex_cloud
bool
Whether the app is running in the reflex cloud environment.
extra_overlay_function
:
str
Extra overlay function to run after the app is built. Formatted such that
from path_0.path_1... import path[-1]
, and calling it with no arguments would work. For example, "reflex.components.moment.moment".
plugins
```

Plugin

List of plugins to use in the app.

Methods

Signature

Description

update_from_env(self) -> 'dict[str, Any]'

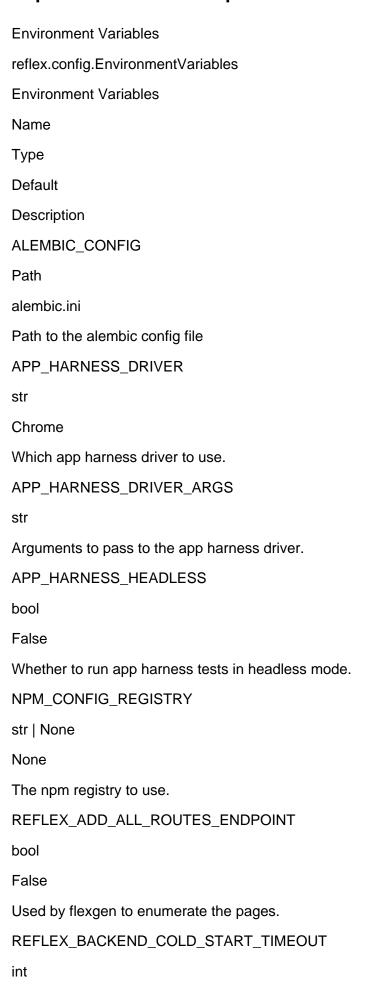
Update the config values based on set environment variables.

If there is a set env_file, it is loaded first.

get_event_namespace(self) -> 'str'

Get the path that the backend Websocket server lists on.

https://reflex.dev/docs/api-reference/environment-variables



The timeout for the backend to do a cold start in seconds.

REFLEX_BACKEND_ONLY

bool

False

Whether to run the backend only. Exclusive with REFLEX_FRONTEND_ONLY.

REFLEX_BACKEND_PORT

int | None

None

The port to run the backend on.

REFLEX_BUILD_BACKEND

str

https://flexgen-prod-flexgen.fly.dev

The reflex.build backend host.

REFLEX_BUILD_FRONTEND

str

https://reflex.build

The reflex.build frontend host.

REFLEX_CHECK_LATEST_VERSION

bool

True

Whether to check for outdated package versions.

__REFLEX_COMPILE_CONTEXT

CompileContext

CompileContext.UNDEFINED

Indicate the current command that was invoked in the reflex CLI.

REFLEX_COMPILE_EXECUTOR

reflex.environment.ExecutorType | None

None

REFLEX_COMPILE_PROCESSES

int | None

None

Whether to use separate processes to compile the frontend and how many. If not set, defaults to thread executor.

int | None None Whether to use separate threads to compile the frontend and how many. Defaults to $min(32, os.cpu_count() + 4)$ REFLEX_DIR Path /home/runner/.local/share/reflex The directory to store reflex dependencies. REFLEX_DOES_BACKEND_COLD_START bool False Enables different behavior for when the backend would do a cold start if it was inactive. REFLEX ENABLE FULL LOGGING bool False Enable full logging of debug messages to reflex user directory. REFLEX_ENV_MODE Env Env.DEV This env var stores the execution mode of the app REFLEX_FRONTEND_ONLY bool **False** Whether to run the frontend only. Exclusive with REFLEX_BACKEND_ONLY. REFLEX FRONTEND PORT int | None None The port to run the frontend on. REFLEX_HOT_RELOAD_EXCLUDE_PATHS list П

Paths to exclude from the hot reload. Takes precedence over include paths. Separated by a colon.

REFLEX_COMPILE_THREADS

REFLEX_HOT_RELOAD_INCLUDE_PATHS	
list	
Additional paths to include in the hot reload. Separated by a colon.	
REFLEX_HTTP_CLIENT_BIND_ADDRESS	
str None	
None	
The address to bind the HTTP client to. You can set this to "::" to enable IPv6.	
REFLEX_IGNORE_REDIS_CONFIG_ERROR	
bool	
False	
Whether to ignore the redis config error. Some redis servers only allow out-of-band configuration.	
REFLEX_LOG_FILE	
pathlib.Path None	
None	
The path to the reflex log file. If not set, the log file will be stored in the reflex user directory.	
REFLEX_PERF_MODE	
PerformanceMode	
PerformanceMode.WARN	
In which performance mode to run the app.	
REFLEX_PERSIST_WEB_DIR	
bool	
False	
Whether to skip purging the web directory in dev mode.	
REFLEX_SKIP_COMPILE	
bool	
False	
If this env var is set to "yes", App.compile will be a no-op	
REFLEX_SOCKET_INTERVAL	
int	
25	
The interval to send a ping to the websocket server in seconds.	
REFLEX SOCKET MAX HTTP BLIFFER SIZE	

int

1000000 Maximum size of the message in the websocket server in bytes. REFLEX_SOCKET_TIMEOUT int 120 The timeout to wait for a pong from the websocket server in seconds. REFLEX_STATES_WORKDIR Path .states The working directory for the states directory. REFLEX_STATE_SIZE_LIMIT int 1000 The maximum size of the reflex state in kilobytes. REFLEX_STRICT_HOT_RELOAD bool False Whether to run Granian in a spawn process. This enables Reflex to pick up on environment variable changes between hot reloads. REFLEX_UPLOADED_FILES_DIR Path uploaded_files The directory to store uploaded files. REFLEX_USE_GRANIAN bool False Whether to use Granian for the backend. By default, the backend uses Uvicorn if available. REFLEX_USE_NPM

bool

False

Whether to use npm over bun to install and run the frontend.

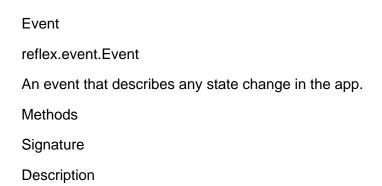
REFLEX_USE_SYSTEM_BUN

bool

False

Whether to use the system installed bun. If set to false, bun will be bundled with the app.
REFLEX_USE_TURBOPACK
bool
False
Whether to use the turbopack bundler.
REFLEX_WEB_WORKDIR
Path
.web
The working directory for the frontend directory.
SQLALCHEMY_ECHO
bool
False
Whether to print the SQL queries if the log level is INFO or lower.
SQLALCHEMY_POOL_PRE_PING
bool
True
Whether to check db connections before using them.
SSL_NO_VERIFY
bool
False
Disable SSL verification for HTTPX requests.

https://reflex.dev/docs/api-reference/event



https://reflex.dev/docs/api-reference/event-triggers

Event	Tric	ide	rs
	1110	ıwu	,, ,

Components can modify the state based on user events such as clicking a button or entering text in a field.

These events are triggered by event triggers.

Event triggers are component specific and are listed in the documentation for each component.

Component Lifecycle Events

Reflex components have lifecycle events like

on_mount

and

on_unmount

that allow you to execute code at specific points in a component's existence. These events are crucial for initializing data, cleaning up resources, and creating dynamic user interfaces.

When Lifecycle Events Are Activated

on_mount

: This event is triggered immediately after a component is rendered and attached to the DOM. It fires:

When a page containing the component is first loaded

When a component is conditionally rendered (appears after being hidden)

When navigating to a page containing the component using internal navigation

It does NOT fire when the page is refreshed or when following external links

on_unmount

: This event is triggered just before a component is removed from the DOM. It fires:

When navigating away from a page containing the component using internal navigation

When a component is conditionally removed from the DOM (e.g., via a condition that hides it)

It does NOT fire when refreshing the page, closing the browser tab, or following external links

Page Load Events

In addition to component lifecycle events, Reflex also provides page-level events like

on_load

that are triggered when a page loads. The

on_load

event is useful for:

Fetching data when a page first loads

Checking authentication status

Initializing page-specific state

Setting default values for cookies or browser storage

You can specify an event handler to run when the page loads using the

on load

parameter in the

@rx.page

decorator or

app.add_page()

method:

This is particularly useful for authentication checks:

For more details on page load events, see the

page load events documentation

.

Event Reference

on_focus

The on_focus event handler is called when the element (or some element inside of it) receives focus. For example, it's called when the user clicks on a text input.

on_blur

The on_blur event handler is called when focus has left the element (or left some element inside of it). For example, it's called when the user clicks outside of a focused text input.

on_change

The on_change event handler is called when the value of an element has changed. For example, it's called when the user types into a text input each keystroke triggers the on change.

on_click

The on_click event handler is called when the user clicks on an element. For example, it's called when the user clicks on a button.

Change Me!

on context menu

The on_context_menu event handler is called when the user right-clicks on an element. For example, it's called when the user right-clicks on a button.

Change Me!

on_double_click

The on_double_click event handler is called when the user double-clicks on an element. For example, it's called when the user double-clicks on a button.

Change Me!

on mount

The on_mount event handler is called after the component is rendered on the page. It is similar to a page on_load event, although it does not necessarily fire when navigating between pages. This event is particularly useful for initializing data, making API calls, or setting up component-specific state when a component first appears.

Component Lifecycle Demo

on_unmount

The on_unmount event handler is called after removing the component from the page. However, on_unmount will only be called for internal navigation, not when following external links or refreshing the page. This event is useful for cleaning up resources, saving state, or performing cleanup operations before a component is removed from the DOM.

Unmount Demo

Resource active

Navigate Away (Triggers Unmount)

on_mouse_up

The on_mouse_up event handler is called when the user releases a mouse button on an element. For example, it's called when the user releases the left mouse button on a button.

Change Me!

on mouse down

The on_mouse_down event handler is called when the user presses a mouse button on an element. For example, it's called when the user presses the left mouse button on a button.

Change Me!

on_mouse_enter

The on_mouse_enter event handler is called when the user's mouse enters an element. For example, it's called when the user's mouse enters a button.

Change Me!

on mouse leave

The on_mouse_leave event handler is called when the user's mouse leaves an element. For example, it's called when the user's mouse leaves a button.

Change Me!

on_mouse_move

The on_mouse_move event handler is called when the user moves the mouse over an element. For example, it's called when the user moves the mouse over a button.

Change Me!

on_mouse_out

The on_mouse_out event handler is called when the user's mouse leaves an element. For example, it's called when the user's mouse leaves a button.

Change Me!

on_mouse_over

The on_mouse_over event handler is called when the user's mouse enters an element. For example, it's called when the user's mouse enters a button.

Change Me!

on_scroll

The on_scroll event handler is called when the user scrolls the page. For example, it's called when the user scrolls the page down.

Scroll to make the text below change.

Change Me!

Scroll to make the text above change.

https://reflex.dev/docs/api-reference/eventhandler

Eventhandler
reflex.event.EventHandler
An event handler responds to an event to update the state.
Methods
Signature
Description
get_parameters(self) -> collections.abc.Mapping[str, inspect.Parameter]
Get the parameters of the function.

https://reflex.dev/docs/api-reference/eventspec

Eventspec

reflex.event.EventSpec

An event specification.

Whereas an Event object is passed during runtime, a spec is used

during compile time to outline the structure of an event.

Methods

Signature

Description

with_args(self, args: tuple[tuple[reflex.vars.base.Var, reflex.vars.base.Var], ...]) -> 'EventSpec'

Copy the event spec, with updated args.

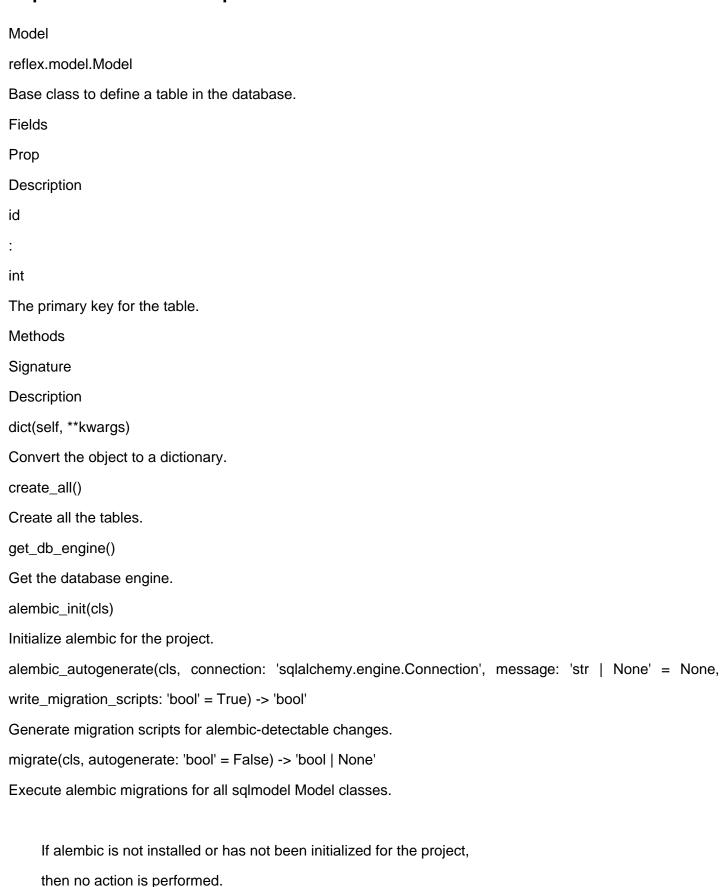
add_args(self, *args: reflex.vars.base.Var) -> 'EventSpec'

Add arguments to the event spec.

https://reflex.dev/docs/api-reference/importvar

Importvar
reflex.utils.imports.ImportVar
An import var.
Methods
Signature
Description

https://reflex.dev/docs/api-reference/model



If there are no revisions currently tracked by alembic, then

an initial revision will be created based on sqlmodel metadata.

If models in the app have changed in incompatible ways that alembic cannot automatically generate revisions for, the app may not be able to start up until migration scripts have been corrected by hand.

select(cls)

Select rows from the table.

https://reflex.dev/docs/api-reference/special-events

Special Events

Reflex includes a set of built-in special events that can be utilized as event triggers

or returned from event handlers in your applications. These events enhance interactivity and user experience.

Below are the special events available in Reflex, along with explanations of their functionality:

rx.console_log

Perform a console.log in the browser's console.

Log

When triggered, this event logs a specified message to the browser's developer console.

It's useful for debugging and monitoring the behavior of your application.

rx.scroll_to

scroll to an element in the page

Scroll to download button

When this is triggered, it scrolls to an element passed by id as parameter. Click on button to scroll to download button (rx.download section) at the bottom of the page

rx.redirect

Redirect the user to a new path within the application.

Parameters

path

: The destination path or URL to which the user should be redirected.

external

: If set to True, the redirection will open in a new tab. Defaults to

False

open in tab

open in new tab

When this event is triggered, it navigates the user to a different page or location within your Reflex application.

By default, the redirection occurs in the same tab. However, if you set the external parameter to True, the redirection

will open in a new tab or window, providing a seamless user experience.

This event can also be run from an event handler in State. It is necessary to

return

the
rx.redirect()
•
Change page in State
rx.set_clipboard
Set the specified text content to the clipboard.
Copy "Hello World" to clipboard
This event allows you to copy a given text or content to the user's clipboard.
It's handy when you want to provide a "Copy to Clipboard" feature in your application,
allowing users to easily copy information to paste elsewhere.
rx.set_value
Set the value of a specified reference element.
Erase
With this event, you can modify the value of a particular HTML element, typically an input field or another
form element.
rx.window_alert
Create a window alert in the browser.
Alert
rx.download
Download a file at a given path.
Parameters:
url
: The URL of the file to be downloaded.
data
: The data to be downloaded. Should be
str
or
bytes
,
data:
URI,
PIL.Image
, or any state Var (to be converted to JSON).
filename

: The desired filename of the downloaded file.
url
and
data
args are mutually exclusive, and at least one of them must be provided.
Download

https://reflex.dev/docs/api-reference/state



reflex.state.State

The app Base State.

Methods

Signature

Description

set_is_hydrated(*args: Any, **kwargs: Any) -> 'EventSpec'

An event handler responds to an event to update the state.

setvar(*args: 'Any') -> 'EventSpec'

A special event handler to wrap setvar functionality.

https://reflex.dev/docs/api-reference/statemanager

Statemanager

reflex.istate.manager.StateManager

A class to manage many client states.

Methods

Signature

Description

create(cls, state: type[reflex.state.BaseState])

Create a new state manager.

get_state(self, token: str) -> reflex.state.BaseState

Get the state for a token.

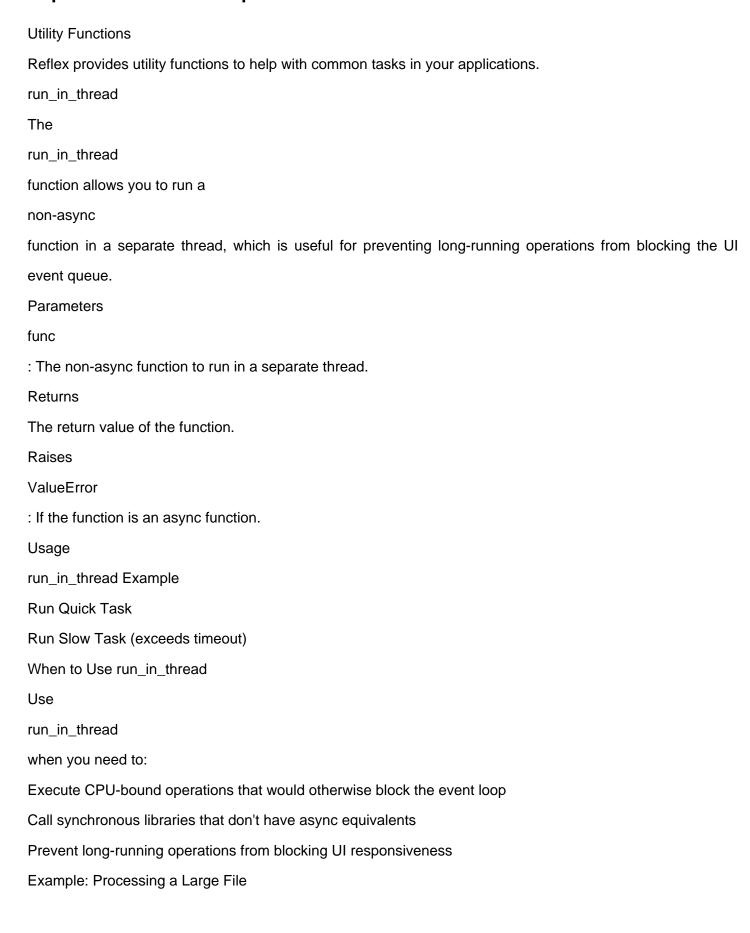
set_state(self, token: str, state: reflex.state.BaseState)

Set the state for a token.

modify_state(self, token: str) -> collections.abc.AsyncIterator[reflex.state.BaseState]

Modify the state for a token while holding exclusive lock.

https://reflex.dev/docs/api-reference/utils



https://reflex.dev/docs/api-reference/var

Var

reflex.vars.base.Var

Base class for immutable vars.

Methods

Signature

Description

equals(self, other: 'Var') -> 'bool'

Check if two vars are equal.

create(cls, value: 'OTHER_VAR_TYPE', _var_data: 'VarData | None' = None) -> 'Var[OTHER_VAR_TYPE]'

Create a var from a value.

to(self, output: 'type[OUTPUT] | types.GenericType', var_type: 'types.GenericType | None' = None) -> 'Var'

Convert the var to a different type.

guess_type(self) -> 'Var'

Guesses the type of the variable based on its `_var_type` attribute.

bool(self) -> 'BooleanVar'

Convert the var to a boolean.

is_none(self) -> 'BooleanVar'

Check if the var is None.

is not none(self) -> 'BooleanVar'

Check if the var is not None.

to_string(self, use_json: 'bool' = True) -> 'StringVar'

Convert the var to a string.

js_type(self) -> 'StringVar'

Returns the javascript type of the object.

This method uses the `typeof` function from the `FunctionStringVar` class

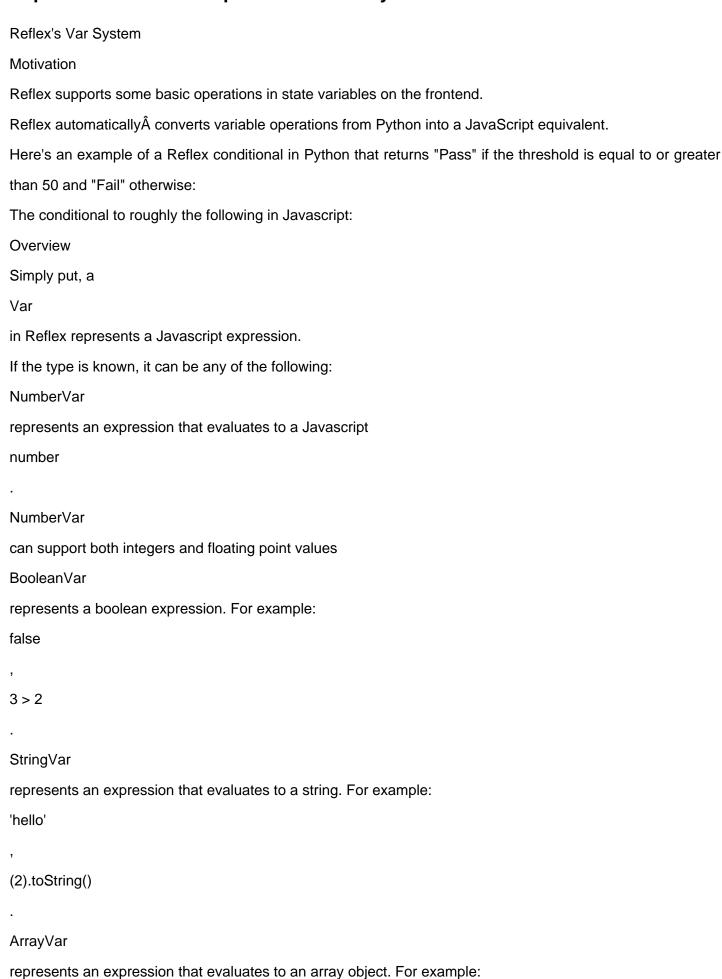
to determine the type of the object.

range(cls, first_endpoint: 'int | NumberVar', second_endpoint: 'int | NumberVar | None' = None, step: 'int |

NumberVar | None' = None) -> 'ArrayVar[Sequence[int]]'

Create a range of numbers.

https://reflex.dev/docs/api-reference/var-system



```
[1, 2, 3]
'words'.split()
ObjectVar
represents an expression that evaluates to an object. For example:
{a: 2, b: 3}
\{deeply: \{nested: {value: false}}}
NoneVar
represent null values. These can be either
undefined
or
null
Creating Vars
State fields are converted to
Var
by default. Additionally, you can create a
Var
from Python values using
rx.Var.create()
If you want to explicitly create a
Var
from a raw Javascript string, you can instantiate
rx.Var
directly:
In the example above,
.guess_type()
will attempt to downcast from a generic
Var
type into
```



in the' factorial' function, we instantiate an array using the range function, and pass this array to multiply_array_values

https://reflex.dev/docs/api-routes/overview

API Transformer

In addition to your frontend app, Reflex uses a FastAPI backend to serve your app. The API transformer feature allows you to transform or extend the ASGI app that serves your Reflex application.

Overview

The API transformer provides a way to:

Integrate existing FastAPI or Starlette applications with your Reflex app

Apply middleware or transformations to the ASGI app

Extend your Reflex app with additional API endpoints

This is useful for creating a backend API that can be used for purposes beyond your Reflex app, or for integrating Reflex with existing backend services.

Using API Transformer

You can set the

api_transformer

parameter when initializing your Reflex app:

Types of API Transformers

The

api_transformer

parameter can accept:

A Starlette or FastAPI instance

A callable that takes an ASGIApp and returns an ASGIApp

A sequence of the above

Using a FastAPI or Starlette Instance

When you provide a FastAPI or Starlette instance as the API transformer, Reflex will mount its internal API to your app, allowing you to define additional routes:

Using a Callable Transformer

You can also provide a callable that transforms the ASGI app:

Using Multiple Transformers

You can apply multiple transformers by providing a sequence:

Reserved Routes

Some routes on the backend are reserved for the runtime of Reflex, and should not be overridden unless you know what you are doing.

Ping

localhost:8000/ping/
: You can use this route to check the health of the backend.

The expected return is
"pong"

Event
localhost:8000/_event
: the frontend will use this route to notify the backend that an event occurred.

Overriding this route will break the event communication

Upload
localhost:8000/_upload
: This route is used for the upload of file when using
rx.upload()
.

https://reflex.dev/docs/assets/overview



Shared assets are placed next to your Python file and are linked to the app's external assets directory. This is useful for creating reusable components with their own assets:

You can also specify a subfolder for shared assets:

Shared assets are linked to your app via symlinks.

Favicon

The favicon is the small icon that appears in the browser tab.

You can add a

favicon.ico

file to the

assets/

folder to change the favicon.

https://reflex.dev/docs/assets/upload-and-download-files

Files

In addition to any assets you ship with your app, many web app will often need to receive or send files, whether you want to share media, allow user to import their data, or export some backend data.

In this section, we will cover all you need to know for manipulating files in Reflex.

Assets vs Upload Directory

Before diving into file uploads and downloads, it's important to understand the difference between assets and the upload directory in Reflex:

Feature

Assets

Upload Directory

Purpose

Static files included with your app (images, stylesheets, scripts)

Dynamic files uploaded by users during runtime

Location

assets/

folder or next to Python files (shared assets)

uploaded_files/

directory (configurable)

Access Method

rx.asset()

or direct path reference

rx.get_upload_url()

When to Use

For files that are part of your application's codebase

For files that users upload or generate through your application

Availability

Available at compile time

Available at runtime

For more information about assets, see the

Assets Overview

Download

If you want to let the users of your app download files from your server to their computer, Reflex offer you two
way.
With a regular link
For some basic usage, simply providing the path to your resource in a
rx.link
will work, and clicking the link will download or display the resource.
Download
With
Using the
rx.download
event will always prompt the browser to download the file, even if it could be displayed in the browser.
The
rx.download
event also allows the download to be triggered from another backend event handler.
Download
rx.download
lets you specify a name for the file that will be downloaded, if you want it to be different from the name on the
server side.
Download and Rename
If the data to download is not already available at a known URL, pass the
data
directly to the
rx.download
event from the backend.
Download random numbers
The
data
arg accepts
str
or
bytes
data, a
data:
URI,

https://reflex.dev/docs/authentication/authentication-overview

Authentication Overview

Many apps require authentication to manage users. There are a few different ways to accomplish this in Reflex:

We have solutions that currently exist outside of the core framework:

Local Auth: Uses your own database:

https://github.com/masenf/reflex-local-auth

Google Auth: Uses sign in with Google:

https://github.com/masenf/reflex-google-auth

Captcha: Generates tests that humans can pass but automated systems cannot:

https://github.com/masenf/reflex-google-recaptcha-v2

Magic Link Auth: A passwordless login method that sends a unique, one-time-use URL to a user's email:

https://github.com/masenf/reflex-magic-link-auth

Clerk Auth: A community member wrapped this component and hooked it up in this app:

https://github.com/TimChild/reflex-clerk-api

Guidance for Implementing Authentication

Store sensitive user tokens and information in

backend-only vars

Validate user session and permissions for each event handler that performs an authenticated action and all computed vars or loader events that access private data.

All content that is statically rendered in the frontend (for example, data hardcoded or loaded at compile time in the UI) will be publicly available, even if the page redirects to a login or uses

rx.cond

to hide content.

Only data that originates from state can be truly private and protected.

When using cookies or local storage, a signed JWT can detect and invalidate any local tampering.

More auth documentation on the way. Check back soon!

https://reflex.dev/docs/client-storage/overview

Client-storage

You can use the browser's local storage to persist state between sessions.

This allows user preferences, authentication cookies, other bits of information

to be stored on the client and accessed from different browser tabs.

A client-side storage var looks and acts like a normal

str

var, except the

default value is either

rx.Cookie

or

rx.LocalStorage

depending on where the

value should be stored. The key name will be based on the var name, but this

can be overridden by passing

name="my_custom_name"

as a keyword argument.

For more information see

Browser Storage

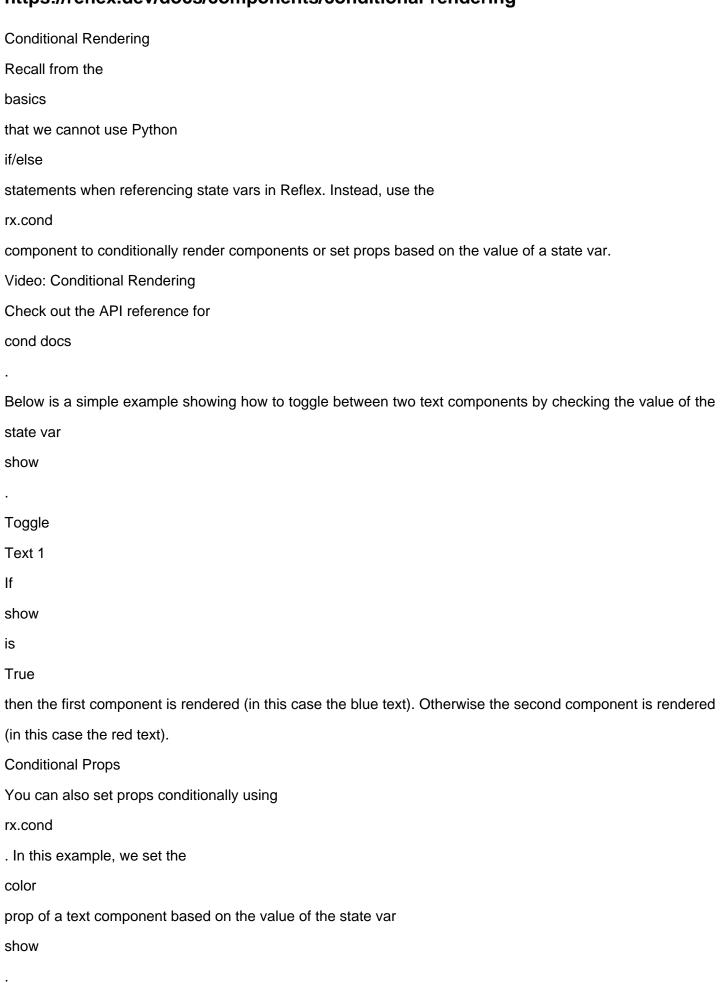
Try entering some values in the text boxes below and then load the page in a separate tab or check the storage section of browser devtools to see the values saved in the browser.

my_cookie

my_local_storage

custom_cookie

https://reflex.dev/docs/components/conditional-rendering



Var Operations
You can use
var operations
with the
cond
component for more complex conditions. See the full
cond reference
for more details.
Multiple Conditional Statements
The
rx.match
component in Reflex provides a powerful alternative to
rx.cond
for handling multiple conditional statements and structural pattern matching. This component allows you to
handle multiple conditions and their associated components in a cleaner and more readable way compared
to nested
rx.cond
structures.
Unknown cat breed selected.

https://reflex.dev/docs/components/html-to-reflex

Convert HTML to Reflex

To convert HTML to Reflex code use this live converter tool:

https://reflex.build/reverse-compiler/

Convert Figma file to Reflex

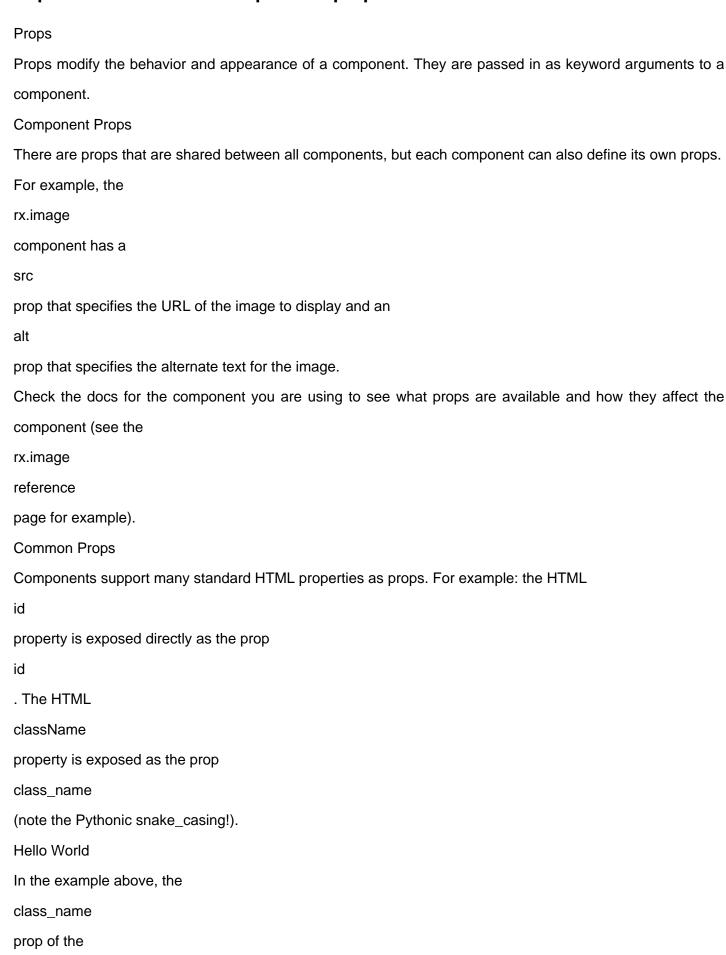
Check out this

Notion doc

for a walk through on how to convert a Figma file into Reflex code.

https://reflex.dev/docs/components/props

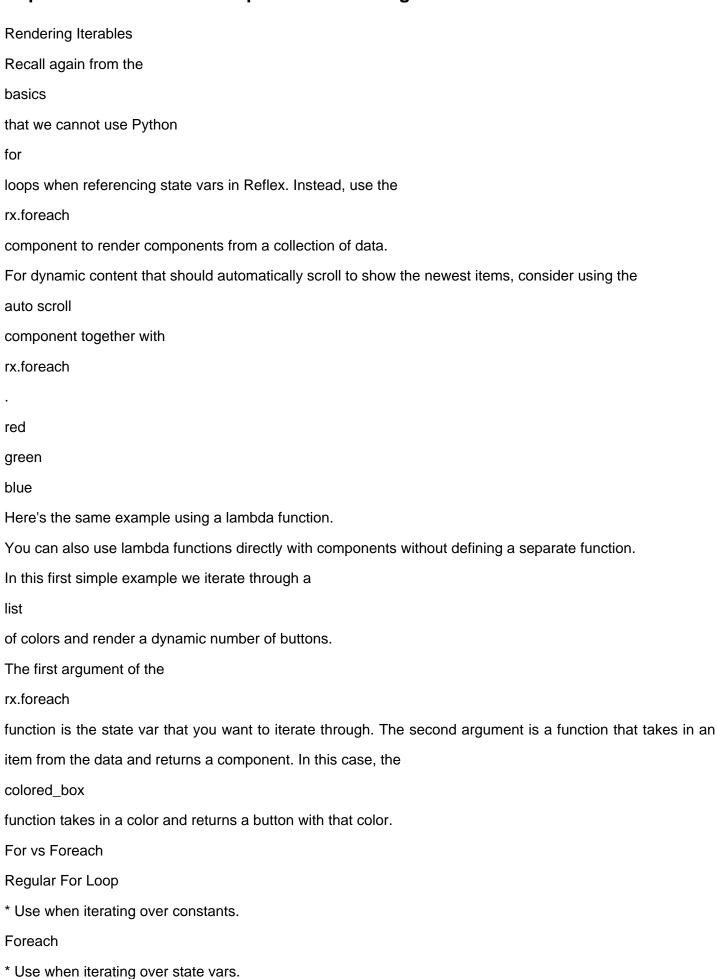
rx.box



component is assigned a list of class names. This means the
rx.box
component will be styled with the CSS classes
class-name-1
and
class-name-2
Style Props
In addition to component-specific props, most built-in components support a full range of style props. You can
use any
CSS property
to style a component.
Fancy Button
See the
styling docs
to learn more about customizing the appearance of your app.
Binding Props to State
Optional: Learn all about
State
first.
Reflex apps define
State
classes that hold variables that can change over time.
State may be modified in response to things like user input like clicking a button, or in response to events like
loading a page.
State vars can be bound to component props, so that the UI always reflects the current state of the app.
Try clicking the badge below to change its color.
Hello World
In this example, the
color_scheme
prop is bound to the
color
state var.
When the

flip_color
event handler is called, the
color
var is updated, and the
color_scheme
prop is updated to match.

https://reflex.dev/docs/components/rendering-iterables



Regular For Loop * Use when iterating over constants. Foreach * Use when iterating over state vars. The above example could have been written using a regular Python for loop, since the data is constant. red green blue However, as soon as you need the data to be dynamic, you must use rx.foreach red green blue Add Render Function The function to render each item can be defined either as a separate function or as a lambda function. In the example below, we define the function colored_box separately and pass it to the rx.foreach function. red green blue Notice that the type annotation for the color parameter in the colored_box function is rx.Var[str] (rather than just

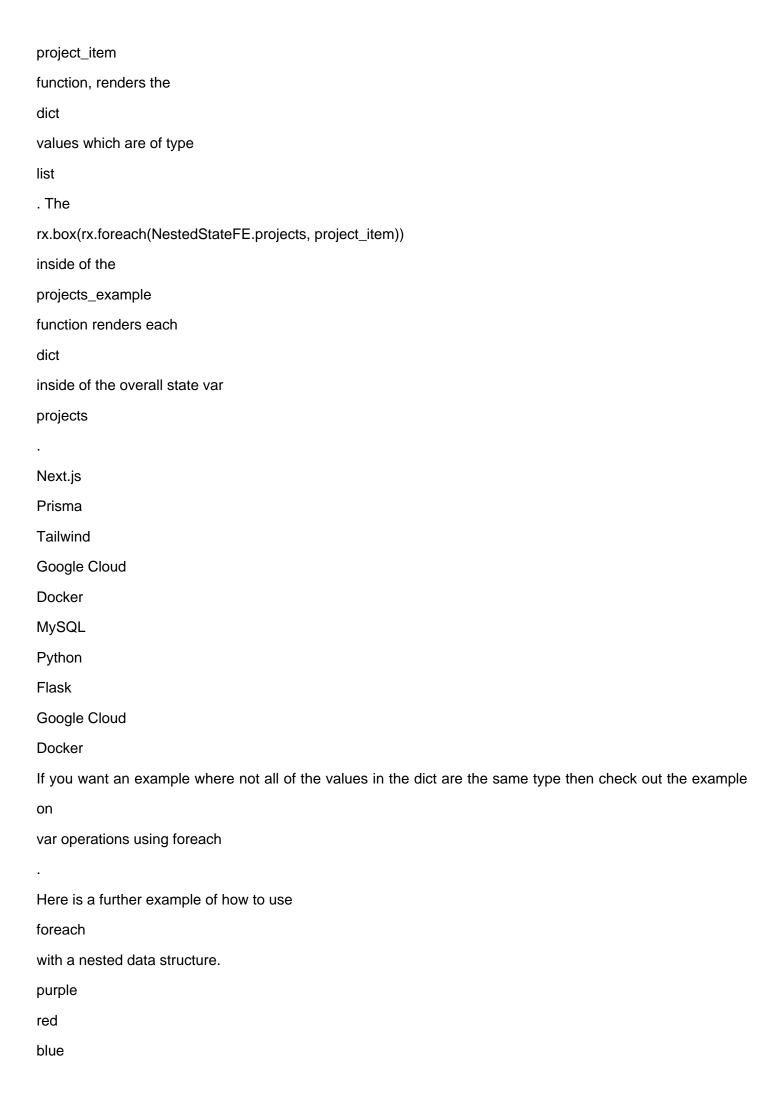
str). This is because the rx.foreach function passes the item as a Var object, which is a wrapper around the actual value. This is what allows us to compile the frontend without knowing the actual value of the state var (which is only known at runtime). **Enumerating Iterables** The function can also take an index as a second argument, meaning that we can enumerate through data as shown in the example below. 1. red 2. green 3. blue Here's the same example using a lambda function. **Iterating Dictionaries** We can iterate through a dict using a foreach . When the dict is passed through to the function that renders each item, it is presented as a list of key-value pairs [("sky", "blue"), ("balloon", "red"), ("grass", "green")] sky balloon grass Dict Type Annotation. Nested examples rx.foreach can be used with nested state vars. Here we use nested

foreach

inside of the

components to render the nested state vars. The

rx.foreach(project["technologies"], get_badge)



orange
yellow
red
green
blue
yellow
Foreach with Cond
We can also use
foreach
with the
cond
component.
In this example we define the function
render_item
. This function takes in an
item
, uses the
cond
to check if the item
is_packed
. If it is packed it returns the
item_name
with a
âœ"
next to it, and if not then it just returns the
item_name
. We use the
foreach
to iterate over all of the items in the
to_do_list
using the
render_item
function.
Sammy's Packing List

Space suit âœ"

Helmet âœ"

Back Pack

https://reflex.dev/docs/custom-components

Custom Components

Reflex has a growing ecosystem of custom components that you can use to build your apps. Below is a list of some of the custom components available for Reflex.

Sort

Package Name

Last Updated

Install Command

Docs

Rows per page

Page 1 of 1

https://reflex.dev/docs/custom-components/command-reference

Command Reference
The custom component commands are under
reflex component
subcommand. To see the list of available commands, run
reflex componenthelp
. To see the manual on a specific command, run
reflex component <command/> help
, for example,
reflex component inithelp
reflex component init
Below is an example of running the
init
command.
The
init
command uses the current enclosing folder name to construct a python package name, typically in the kebal
case. For example, if running init in folder
google_auth
, the package name will be
reflex-google-auth
. The added prefix reduces the chance of name collision on PyPI (the Python Package Index), and it indicates
that the package is a Reflex custom component. The user can override the package name by providing the
package-name
option.
The
init
command creates a set of files and folders prefilled with the package name and other details. During the init
the
custom_component
folder is installed locally in editable mode, so a developer can incrementally develop and test with ease. The

changes in component implementation is automatically reflected where it is used. Below is the folder

structure after the
init
command.
pyproject.toml
The
pyproject.toml
is required for the package to build and be published. It is prefilled with information such as the package
name, version (
0.0.1
), author name and email, homepage URL. By default the
Apache-2.0
license is used, the same as Reflex. If any of this information requires update, the user can edit the file by
hand.
README
The
README.md
file is created with installation instructions, e.g.
pip install reflex-google-auth
, and a brief description of the package. Typically the
README.md
contains usage examples. On PyPI, the
README.md
is rendered as part of the package page.
Custom Components Folder
The
custom_components
folder is where the actual implementation is. Do not worry about this folder name: there is no need to change
it. It is where
pyproject.toml
specifies the source of the python package is. The published package contains the contents inside it,
excluding this folder.
reflex_google_auth
is the top folder for importable code. The
reflex_google_auth/initpy

```
imports everything from the
reflex_google_auth/google_auth.py
. For the user of the package, the import looks like
from reflex_google_auth import ABC, XYZ
reflex_google_auth/google_auth.py
is prefilled with code example and instructions from the
wrapping react guide
Demo App Folder
A demo app is generated inside
google_auth_demo
folder with import statements and example usage of the component. This is a regular Reflex app. Go into this
directory and start using any reflex commands for testing. The user is encouraged to deploy the demo app,
so it can later be included as part of the
Gallery
Help Manual
The help manual is shown when adding the
--help
option to the command.
reflex component publish
To publish to a package index, a user is required to already have an account with them. As of
0.7.5
, Reflex does not handle the publishing process for you. You can do so manually by first running
reflex component build
followed by
twine upload
or
uv publish
or your choice of a publishing utility.
You can then share your build on our website with
reflex component share
```

reflex component build
It is not required to run the
build
command separately before publishing. The
publish
command will build the package if it is not already built. The
build
command is provided for the user's convenience.
The
build
command generates the
.tar.gz
and
.whl
distribution files to be uploaded to the desired package index, for example, PyPI. This command must be run
at the top level of the project where the
pyproject.toml
file is. As a result of a successful build, there is a new
dist
folder with the distribution files.

https://reflex.dev/docs/custom-components/overview

Custom Components Overview

Reflex users create many components of their own: ready to use high level components, or nicely wrapped React components. With

Custom Components

, the community can easily share these components now.

Release

0.4.3

introduces a series of

reflex component

commands that help developers wrap react components, test, and publish them as python packages. As shown in the image below, there are already a few custom components published on PyPI, such as reflex-spline

reflex-webcam

Check out the custom components gallery

here

Prerequisites for Publishing

In order to publish a Python package, an account is required with a python package index, for example, PyPI. The documentation to create accounts and generate API tokens can be found on their websites. For a quick reference, check out our

Prerequisites for Publishing

page.

Steps to Publishing

Follow these steps to publish the custom component as a python package:

reflex component init

: creates a new custom component project from templates.

dev and test: developer implements and tests the custom component.

reflex component build

: builds the package.

twine upload

or

uv publish

: uploads the package to a python package index.

Initialization

First create a new folder for your custom component project, for example

color_picker

. The package name will be

reflex-color-picker

. The prefix

reflex-

is intentionally added for all custom components for easy search on PyPI. If you prefer a particular name for the package, you can either change it manually in the

pyproject.toml

file or add the

--library-name

option in the

reflex component init

command initially.

Run

reflex component init

, and a set of files and folders will be created in the

color_picker

folder. The

pyproject.toml

file is the configuration file for the project. The

custom_components

folder is where the custom component implementation is. The

color_picker_demo

folder is a demo Reflex app that uses the custom component. If this is the first time of creating python packages, it is encouraged to browse through all the files (there are not that many) to understand the structure of the project.

Develop and Test

After finishing the custom component implementation, the user is encouraged to fully test it before publishing.

The generated Reflex demo app

color_picker_demo

is a good place to start. It is a regular Reflex app prefilled with imports and usage of this component. During the init, the

custom_component

folder is installed locally in editable mode, so a developer can incrementally develop and test with ease. The changes in component implementation are automatically reflected in the demo app.

Publish

Once you're ready to publish your package, run

reflex component build

to build the package. The command builds the distribution files if they are not already built. The end result is a dist

folder containing the distribution files. The user does not need to do anything manually with these distribution files.

In order to publish these files as a Python package, you need to use a publishing utility. Any would work, but we recommend either

Twine

or (uv)[

https://docs.astral.sh/uv/guides/package/#publishing-your-package

]. Make sure to keep your package version in pyproject.toml updated.

You can also share your components with the rest of the community at our website using the command reflex component share

. See you there!

https://reflex.dev/docs/custom-components/prerequisites-for-publishing

Python Package Index

In order to publish a Python package, you need to use a publishing utility. Any would work, but we recommend either

Twine

or (uv)[

https://docs.astral.sh/uv/guides/package/#publishing-your-package

].

PyPI

It is straightforward to create accounts and API tokens with PyPI. There is official help on the

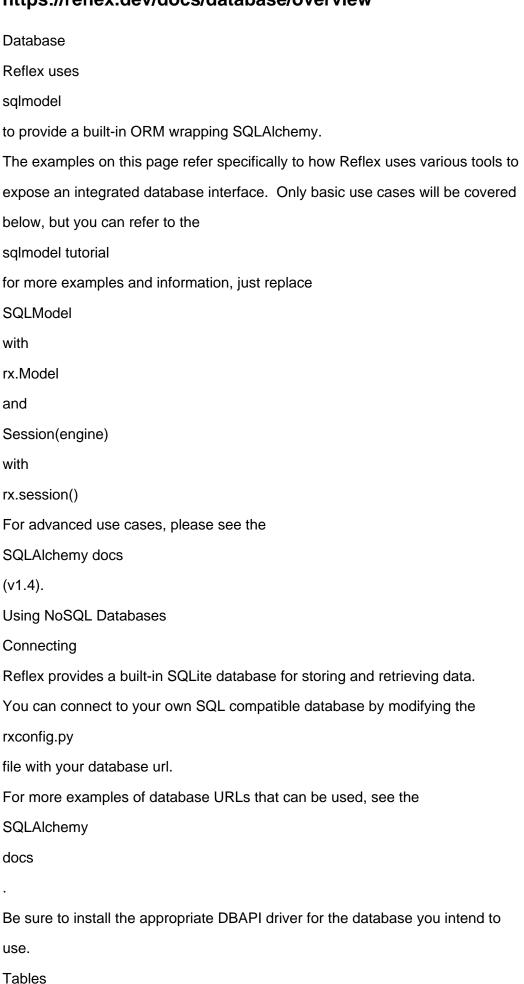
PyPI website

. For a quick reference here, go to the top right corner of the PyPI website and look for the button to register and fill out personal information.

A user can use username and password to authenticate with PyPI when publishing.

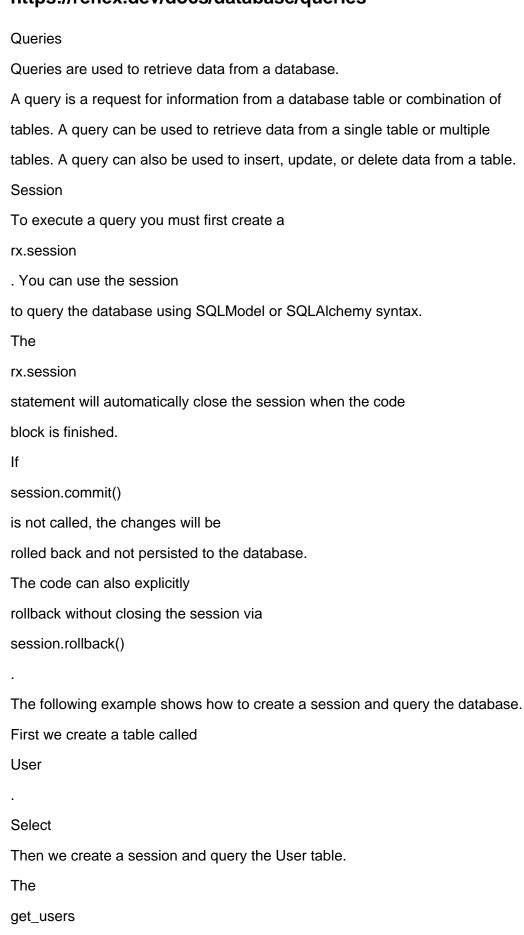
Scroll down to the API tokens section and click on the "Add API token" button. Fill out the form and click "Generate API token".

https://reflex.dev/docs/database/overview



To create a table make a class that inherits from
rx.Model
with and specify
that it is a table.
Migrations
Reflex leverages
alembic
to manage database schema changes.
Before the database feature can be used in a new app you must call
reflex db init
to initialize alembic and create a migration script with the current schema.
After making changes to the schema, use
reflex db makemigrationsmessage 'something changed'
to generate a script in the
alembic/versions
directory that will update the
database schema. It is recommended that generated scripts be inspected before applying them.
Bear in mind that your newest models will not be detected by the
reflex db makemigrations
command unless imported and used somewhere within the application.
The
reflex db migrate
command is used to apply migration scripts to bring the
database up to date. During app startup, if Reflex detects that the current
database schema is not up to date, a warning will be displayed on the console.
Queries
To query the database you can create a
rx.session()
which handles opening and closing the database connection.
You can use normal SQLAlchemy queries to query the database.
Video: Tutorial of Database Model with Forms, Model Field Changes and Migrations, and adding a DateTime
Field

https://reflex.dev/docs/database/queries



method will query the database for all users that contain the

value of the state var

```
name
Insert
Similarly, the
session.add()
method to add a new record to the
database or persist an existing object.
Update
To update the user, first query the database for the object, make the desired
modifications,
.add
the object to the session and finally call
.commit()
Delete
To delete a user, first query the database for the object, then call
.delete()
on the session and finally call
.commit()
ORM Object Lifecycle
The objects returned by queries are bound to the session that created them, and cannot generally
be used outside that session. After adding or updating an object, not all fields are automatically
updated, so accessing certain attributes may trigger additional queries to refresh the object.
To avoid this, the
session.refresh()
method can be used to update the object explicitly and
ensure all fields are up to date before exiting the session.
Now the
self.user
object will have a correct reference to the autogenerated
primary key,
id
, even though this was not provided when the object was created
```

from the form data. lf self.user needs to be modified or used in another query in a new session, it must be added to the session. Adding an object to a session does not necessarily create the object, but rather associates it with a session where it may either be created or updated accordingly. If an ORM object will be referenced and accessed outside of a session, you should call .refresh() on it to avoid stale object exceptions. Using SQL Directly Avoiding SQL is one of the main benefits of using an ORM, but sometimes it is necessary for particularly complex queries, or when using database-specific features. SQLModel exposes the session.execute() method that can be used to execute raw SQL strings. If parameter binding is needed, the query may be wrapped in sqlalchemy.text which allows colon-prefix names to be used as placeholders. Never use string formatting to construct SQL queries, as this may lead to SQL injection vulnerabilities in the app. Async Database Operations Reflex provides an async version of the session function called rx.asession

for asynchronous database operations. This is useful when you need to perform database operations in an async context, such as within async event handlers.

The

rx.asession

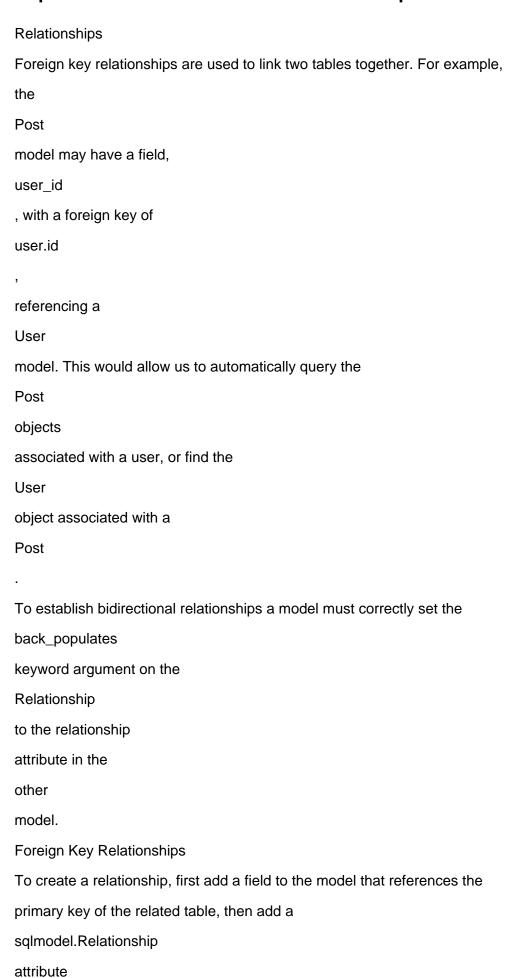
function returns an async SQLAlchemy session that must be used with an async context manager. Most operations against the

asession

must be awaited.

Async Select
The following example shows how to query the database asynchronously:
Async Insert
To add a new record to the database asynchronously:
Async Update
To update a user asynchronously:
Async Delete
To delete a user asynchronously:
Async Refresh
Similar to the regular session, you can refresh an object to ensure all fields are up to date:
Async SQL Execution
You can also execute raw SQL asynchronously:
Important Notes for Async Database Operations

https://reflex.dev/docs/database/relationships



which can be used to access the related objects. Defining relationships like this requires the use of sqlmodel objects as seen in the example. See the **SQLModel Relationship Docs** for more details. Querying Relationships **Inserting Linked Objects** The following example assumes that the flagging user is stored in the state as a User instance and that the post id is provided in the data submitted in the form. How are Relationships Dereferenced? By default, the relationship attributes are in lazy loading or "select" mode, which generates a query on access to the relationship attribute. Lazy loading is generally fine for single object lookups and manipulation, but can be inefficient when accessing many linked objects for serialization purposes. There are several alternative loading mechanisms available that can be set on the relationship object or when performing the query. "joined" or joinload - generates a single query to load all related objects at once. "subquery" or subqueryload

generates a single query to load all related
 objects at once, but uses a subquery to do the join, instead of a join in the main query.

"selectin" or

selectinload

 emits a second (or more) SELECT statement which assembles the primary key identifiers of the parent objects into an IN clause, so that all members of related collections / scalar references are loaded at once by primary key

There are also non-loading mechanisms, "raise" and "noload" which are used to specifically avoid loading a relationship.

Each loading method comes with tradeoffs and some are better suited for different data access patterns.

See

SQLAlchemy: Relationship Loading Techniques

for more detail.

Querying Linked Objects

To query the

Post

table and include all

User

and

Flag

objects up front,

the

.options

interface will be used to specify

selectinload

for the required

relationships. Using this method, the linked objects will be available for rendering in frontend code without additional steps.

The loading methods create new query objects and thus may be linked if the relationship itself has other relationships that need to be loaded. In this example, since

```
Flag
references
User
, the
Flag.user
relationship must be
chain loaded from the
Post.flags
relationship.
Specifying the Loading Mechanism on the Relationship
Alternatively, the loading mechanism can be specified on the relationship by
passing
sa_relationship_kwargs={"lazy": method}
to
sqlmodel.Relationship
which will use the given loading mechanism in all queries by default.
```

https://reflex.dev/docs/database/tables

Tables Tables are database objects that contain all the data in a database. In tables, data is logically organized in a row-and-column format similar to a spreadsheet. Each row represents a unique record, and each column represents a field in the record. Creating a Table To create a table, make a class that inherits from rx.Model The following example shows how to create a table called User The table=True argument tells Reflex to create a table in the database for this class. Primary Key By default, Reflex will create a primary key column called id for each table. However, if an rx.Model defines a different field with primary_key=True , then the default id field will not be created. A table may also redefine id as needed. It is not currently possible to create a table without a primary key.

Advanced Column Types

SQLModel automatically maps basic python types to SQLAlchemy column types, but for more advanced use cases, it is possible to define the column type using sqlalchemy

directly. For example, we can add a last updated timestamp to the post example as a proper

DateTime

field with timezone.

To make the

Post

model more usable on the frontend, a

dict

method may be provided

that converts any fields to a JSON serializable value. In this case, the dict method is overriding the default

datetime

serializer to strip off the microsecond part.

https://reflex.dev/docs/enterprise/built-with-reflex

Built with Reflex Badge

The "Built with Reflex" badge appears in the bottom right corner of apps using reflex-enterprise components.

Removing the Badge

To remove the badge, you need a paid tier:

Cloud

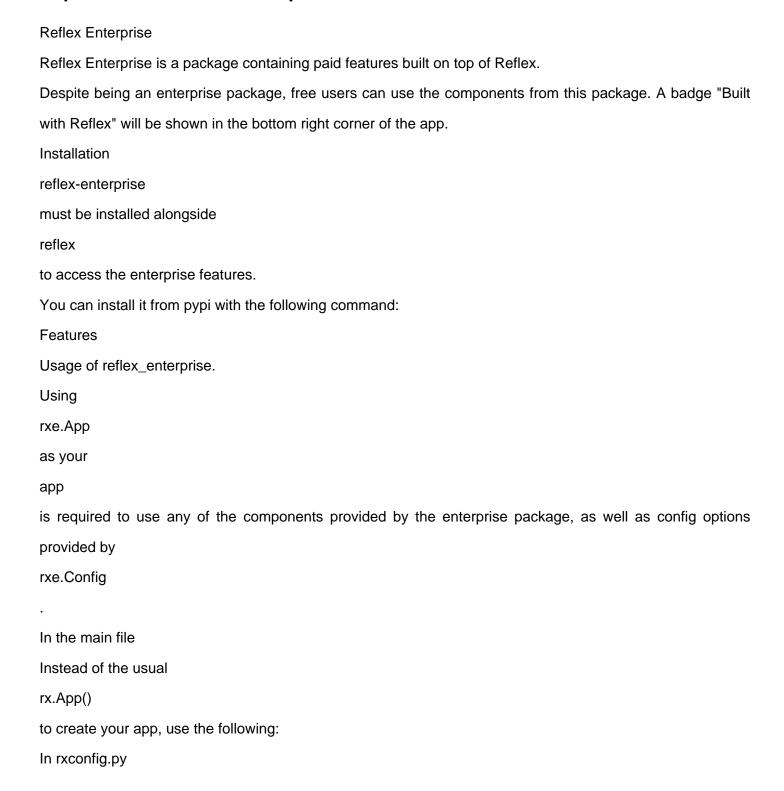
: Pro tier or higher

Self-hosted

: Team tier or higher

Configuration

https://reflex.dev/docs/enterprise/overview



https://reflex.dev/docs/enterprise/single-port-proxy

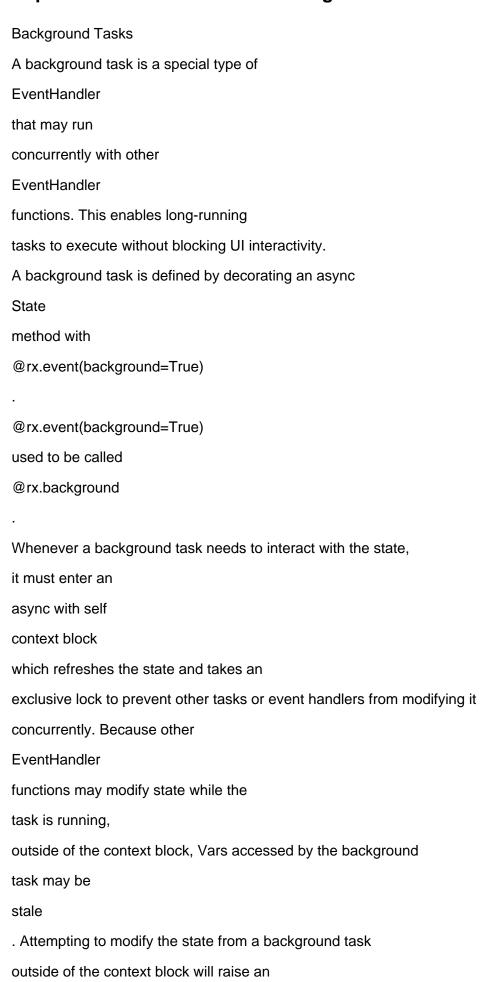
Single Port Proxy

Enable single-port deployment by proxying the backend to the frontend port.

Configuration

This allows your application to run on a single port, which is useful for deployment scenarios where you can only expose one port.

https://reflex.dev/docs/events/background-events



ImmutableStateError exception. In the following example, the my task event handler is decorated with @rx.event(background=True) and increments the counter variable every half second, as long as certain conditions are met. While it is running, the UI remains interactive and continues to process events normally. Background events are similar to simple Task Queues like Celery allowing asynchronous events. 0 / Start Reset Terminating Background Tasks on Page Close or Navigation Sometimes, background tasks may continue running even after the user navigates away from the page or closes the browser tab. To handle such cases, you can check if the websocket associated with the state is disconnected and terminate the background task when necessary. The solution involves checking if the client_token is still valid in the app.event_namespace.token_to_sid mapping. If the session is lost (e.g., the user navigates away or closes the page), the background task will stop. Task Lifecycle When a background task is triggered, it starts immediately, saving a reference to the task in app.background_tasks . When the task completes, it is removed from the set. Multiple instances of the same background task may run concurrently, and the framework makes no attempt to avoid duplicate tasks from starting.

It is up to the developer to ensure that duplicate tasks are not created under

the circumstances that are undesirable. In the example above, the
_n_tasks
backend var is used to control whether
my_task
will enter the increment loop,
or exit early.
Background Task Limitations
Background tasks mostly work like normal
EventHandler
methods, with certain exceptions:
Background tasks must be
async
functions.
Background tasks cannot modify the state outside of an
async with self
context block.
Background tasks may read the state outside of an
async with self
context block, but the value may be stale.
Background tasks may not be directly called from other event handlers or background tasks. Instead use
yield
or
return
to trigger the background task.

https://reflex.dev/docs/events/chaining-events

Chaining events

Calling Event Handlers From Event Handlers

You can call other event handlers from event handlers to keep your code modular. Just use the

self.call_handler

syntax to run another event handler. As always, you can yield within your function to send incremental updates to the frontend.

0

Run

Returning Events From Event Handlers

So far, we have only seen events that are triggered by components. However, an event handler can also return events.

In Reflex, event handlers run synchronously, so only one event handler can run at a time, and the events in the queue will be blocked until the current event handler finishes. The difference between returning an event and calling an event handler is that returning an event will send the event to the frontend and unblock the queue.

Be sure to use the class name

State

(or any substate) rather than

self

when returning events.

Try entering an integer in the input below then clicking out.

1

In this example, we run the

Collatz Conjecture

on a number entered by the user.

When the

on blur

event is triggered, the event handler

start_collatz

is called. It sets the initial count, then calls

run_step

which runs until the count reaches

.

https://reflex.dev/docs/events/decentralized-event-handlers

Decentralized Event Handlers
Overview
Decentralized event handlers allow you to define event handlers outside of state classes, providing more
flexible code organization. This feature was introduced in Reflex v0.7.10 and enables a more modular
approach to event handling.
With decentralized event handlers, you can:
Organize event handlers by feature rather than by state class
Separate UI logic from state management
Create more maintainable and scalable applications
Basic Usage
To create a decentralized event handler, use the
@rx.event
decorator on a function that takes a state instance as its first parameter:
Count: 0
Increment by 1
Increment by 5
Increment by 10
In this example:
We define a
MyState
class with a
count
variable
We create a decentralized event handler
increment
that takes a
MyState
instance as its first parameter
We use the event handler in buttons, passing different amounts to increment by
Compared to Traditional Event Handlers
Here's a comparison between traditional event handlers defined within state classes and decentralized event

handlers:

Key differences: Traditional event handlers use self to reference the state instance Decentralized event handlers explicitly take a state instance as the first parameter Both approaches use the same syntax for triggering events in components Both can be decorated with @rx.event respectively **Best Practices** When to Use Decentralized Event Handlers Decentralized event handlers are particularly useful in these scenarios: Large applications with many event handlers that benefit from better organization Feature-based organization where you want to group related event handlers together Separation of concerns when you want to keep state definitions clean and focused Type Annotations Always use proper type annotations for your state parameter and any additional parameters: Naming Conventions Follow these naming conventions for clarity: Use descriptive names that indicate the action being performed Use the state class name as the type annotation for the first parameter Name the state parameter consistently across your codebase (e.g., always use state or the first letter of the state class) Organization Consider these approaches for organizing decentralized event handlers: Group related event handlers in the same file Place event handlers near the state classes they modify For larger applications, create a dedicated events directory with files organized by feature

Combining with Other Event Features

Decentralized event handlers work seamlessly with other Reflex event features:

https://reflex.dev/docs/events/event-actions

Event Actions

In Reflex, an event action is a special behavior that occurs during or after processing an event on the frontend.

Event actions can modify how the browser handles DOM events or throttle and debounce events before they are processed by the backend.

An event action is specified by accessing attributes and methods present on all EventHandlers and EventSpecs.

DOM Event Propagation

Added in v0.3.2

prevent_default

The

.prevent_default

action prevents the default behavior of the browser for

the action. This action can be added to any existing event, or it can be used on its own by specifying

rx.prevent_default

as an event handler.

A common use case for this is to prevent navigation when clicking a link.

This Link Does Nothing

The value is false

Toggle Value

stop_propagation

The

.stop_propagation

action stops the event from propagating to parent elements.

This action is often used when a clickable element contains nested buttons that should not trigger the parent element's click event.

In the following example, the first button uses

.stop_propagation

to prevent

the click event from propagating to the outer vstack. The second button does not

use

.stop_propagation
, so the click event will also be handled by the on_click
attached to the outer vstack.
btn1 - Stop Propagation
btn2 - Normal Propagation
Reset
Throttling and Debounce
Added in v0.5.0
For events that are fired frequently, it can be useful to throttle or debounce
them to avoid network latency and improve performance. These actions both take a
single argument which specifies the delay time in milliseconds.
throttle
The
.throttle
action limits the number of times an event is processed within a
a given time period. It is useful for
on_scroll
and
on_mouse_move
events which are
fired very frequently, causing lag when handling them in the backend.
Throttled events are discarded.
In the following example, the
on_scroll
event is throttled to only fire every half second.
Scroll Me
Item 0
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6

Item 7

Item 8

Item 9

Item 10

Item 11

Item 12

Item 13

Item 14

Item 15

Item 16

Item 17

Item 18

Item 19

Item 20

Item 21

Item 22

Item 23

Item 24

Item 25

Item 26

Item 27

Item 28

Item 29

Item 30

Item 31

Item 32

Item 33

Item 34

Item 35

Item 36

Item 37

Item 38

Item 39

Item 40

Item 41

Item 42

Item 43

Item 44

Item 45

Item 46

Item 47

Item 48

Item 49

Item 50

Item 51

Item 52

Item 53

Item 54

Item 55 Item 56

Item 57

Item 58

Item 59

Item 60

Item 61

Item 62

Item 63

Item 64

Item 65

Item 66

Item 67

Item 68

Item 69

Item 70

Item 71

Item 72

Item 73

Item 74

Item 75

Item 76
Item 77
Item 78
Item 79
Item 80
Item 81
Item 82
Item 83
Item 84
Item 85
Item 86
Item 87
Item 88
Item 89
Item 90
Item 91
Item 92
Item 93
Item 94
Item 95
Item 96
Item 97
Item 98
Item 99
Last Scroll Event:
Event Actions are Chainable
debounce
The
.debounce
action delays the processing of an event until the specified
timeout occurs. If another event is triggered during the timeout, the timer is
reset and the original event is discarded.
Debounce is useful for handling the final result of a series of events, such as
moving a slider.

Debounced events are discarded.
In the following example, the slider's
on_change
handler,
update_value
, is
only triggered on the backend when the slider value has not changed for half a
second.
Settled Value: 50
Why set key on the slider?
Temporal Events
Added in
v0.6.6
temporal
The
.temporal
action prevents events from being queued when the backend is down.
This is useful for non-existed events whose you do not want them to pile up if there is
This is useful for non-critical events where you do not want them to pile up if there is
a temporary connection issue.
a temporary connection issue.
a temporary connection issue. Temporal events are discarded when the backend is down.
a temporary connection issue. Temporal events are discarded when the backend is down. In the following example, the
a temporary connection issue. Temporal events are discarded when the backend is down. In the following example, the rx.moment
a temporary connection issue. Temporal events are discarded when the backend is down. In the following example, the rx.moment component with
a temporary connection issue. Temporal events are discarded when the backend is down. In the following example, the rx.moment component with interval
a temporary connection issue. Temporal events are discarded when the backend is down. In the following example, the rx.moment component with interval and
a temporary connection issue. Temporal events are discarded when the backend is down. In the following example, the rx.moment component with interval and on_change
a temporary connection issue. Temporal events are discarded when the backend is down. In the following example, the rx.moment component with interval and on_change uses
a temporary connection issue. Temporal events are discarded when the backend is down. In the following example, the rx.moment component with interval and on_change uses .temporal
a temporary connection issue. Temporal events are discarded when the backend is down. In the following example, the rx.moment component with interval and on_change uses .temporal to

https://reflex.dev/docs/events/event-arguments

Event Arguments

The event handler signature needs to match the event trigger definition argument count. If the event handler takes two arguments, the event trigger must be able to provide two arguments.

Here	is a	simple	examp	le:
		01111010	O/tollip	. •

50

The event trigger here is

on_value_commit

and it is called when the value changes at the end of an interaction. This event trigger passes one argument, which is the value of the slider. The event handler which is triggered by the event trigger must therefore take one argument, which is

value

here.

Here is a form example:

Checked

Submit

Results

{}

In this example the event trigger is the

on submit

event of the form. The event handler is

handle_submit

. The

on submit

event trigger passes one argument, the form data as a dictionary, to the

handle_submit

event handler. The

handle submit

event handler must take one argument because the

on_submit

event trigger passes one argument.

When the number of args accepted by an EventHandler differs from that provided by the event trigger, an EventHandlerArgMismatch

error will be raised.

Pass Additional Arguments to Event Handlers

In some use cases, you want to pass additional arguments to your event handlers. To do this you can bind an event trigger to a lambda, which can call your event handler with the arguments you want.

Try typing a color in an input below and clicking away from it to change the color of the input.

In this case, in we want to pass two arguments to the event handler

change_color

, the color and the index of the color to change.

The

on_blur

event trigger passes the text of the input as an argument to the lambda, and the lambda calls the change_color

event handler with the text and the index of the input.

When the number of args accepted by a lambda differs from that provided by the event trigger, an

EventFnArgMismatch

error will be raised.

Event Handler Parameters should provide type annotations.

Events with Partial Arguments (Advanced)

Added in v0.5.0

Event arguments in Reflex are passed positionally. Any additional arguments not passed to an EventHandler will be filled in by the event trigger when it is fired.

The following two code samples are equivalent:

https://reflex.dev/docs/events/events-overview

Events Overview

Events are composed of two parts: Event Triggers and Event Handlers.

Events Handlers

are how the State of a Reflex application is updated. They are triggered by user interactions with the UI, such as clicking a button or hovering over an element. Events can also be triggered by the page loading or by other events.

Event triggers

are component props that create an event to be sent to an event handler.

Each component supports a set of events triggers. They are described in each

component's documentation

in the event trigger section.

Example

Lets take a look at an example below. Try mousing over the heading to change the word.

Welcome

In this example, the heading component has the

event trigger

on_mouse_over

Whenever the user hovers over the heading, the

next_word

event handler

will be called to cycle the word. Once the handler returns, the UI will be updated to reflect the new state.

Adding the

@rx.event

decorator above the event handler is strongly recommended. This decorator enables proper static type checking, which ensures event handlers receive the correct number and types of arguments.

What's in this section?

In the event section of the documentation, you will explore the different types of events supported by Reflex, along with the different ways to call them.

https://reflex.dev/docs/events/page-load-events

Page Load Events

You can also specify a function to run when the page loads. This can be useful for fetching data once vs on every render or state change.

In this example, we fetch data when the page loads:

Another example would be checking if the user is authenticated when the page loads. If the user is not authenticated, we redirect them to the login page. If they are authenticated, we don't do anything, letting them access the page. This

on_load

event would be placed on every page that requires authentication to access.

https://reflex.dev/docs/events/setters

Setters

Every base var has a built-in event handler to set it's value for convenience, called set_VARNAME

.

Say you wanted to change the value of the select component. You could write your own event handler to do this:

1

Or you could could use a built-in setter for conciseness.

1

In this example, the setter for

selected

is

set_selected

. Both of these examples are equivalent.

Setters are a great way to make your code more concise. But if you want to do something more complicated, you can always write your own function in the state.

https://reflex.dev/docs/events/special-events



Reflex also has built-in special events can be found in the reference

For example, an event handler can trigger an alert on the browser.

Alert

Special events can also be triggered directly in the UI by attaching them to an event trigger.

https://reflex.dev/docs/events/yield-events

Yielding Updates

A regular event handler will send a

StateUpdate

when it has finished running. This works fine for basic event, but sometimes we need more complex logic. To update the UI multiple times in an event handler, we can

yield

when we want to send an update.

To do so, we can use the Python keyword

yield

. For every yield inside the function, a

StateUpdate

will be sent to the frontend with the changes up to this point in the execution of the event handler.

This example below shows how to yield 100 updates to the UI.

0

Start

Here is another example of yielding multiple updates with a loading icon.

0

Video: Asyncio with Yield

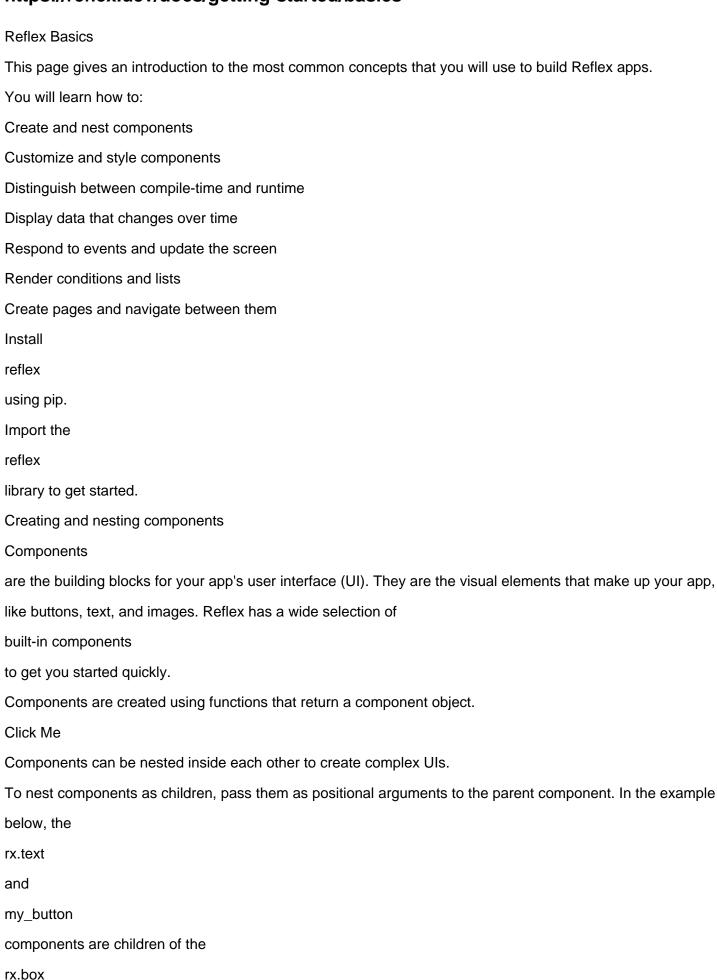
Yielding Other Events

Events can also yield other events. This is useful when you want to chain events together. To do this, you can yield the event handler function itself.

Reference other Event Handler via class

0

https://reflex.dev/docs/getting-started/basics



component.
This is a page
Click Me
You can also use any base HTML element through the
rx.el
namespace. This allows you to use standard HTML elements directly in your Reflex app when you need
more control or when a specific component isn't available in the Reflex component library.
Use base html!
If you need a component not provided by Reflex, you can check the
3rd party ecosystem
or
wrap your own React component
•
Customizing and styling components
Components can be customized using
props
, which are passed in as keyword arguments to the component function.
Each component has props that are specific to that component. Check the docs for the component you are
using to see what props are available.
In addition to component-specific props, components can also be styled using CSS properties passed as
props.
Click Me
Use the
snake_case
version of the CSS property name as the prop name.
See the
styling guide
for more information on how to style components
In summary, components are made up of children and props.
Children
- Text or other Reflex components nested inside a component.
- Passed as **positional arguments**.
Props
- Attributes that affect the behavior and appearance of a component.

- Passed as **keyword arguments**. Children - Text or other Reflex components nested inside a component. - Passed as **positional arguments**. **Props** - Attributes that affect the behavior and appearance of a component. - Passed as **keyword arguments**. Displaying data that changes over time Apps need to store and display data that changes over time. Reflex handles this through State , which is a Python class that stores variables that can change when the app is running, as well as the functions that can change those variables. To define a state class, subclass rx.State and define fields that store the state of your app. The state variables (vars) should have a type annotation, and can be initialized with a default value. Referencing state vars in components To reference a state var in a component, you can pass it as a child or prop. The component will automatically update when the state changes. Vars are referenced through class attributes on your state class. For example, to reference the count var in a component, use MyState.count Count: Vars can be referenced in multiple components, and will automatically update when the state changes. Responding to events and updating the screen So far, we've defined state vars but we haven't shown how to change them. All state changes are handled through functions in the state class, called event handlers Event handlers are the ONLY way to change state in Reflex.

Components have special props, such as

on click

, called event triggers that can be used to make components interactive. Event triggers connect components to event handlers, which update the state.

0

Increment

When an event trigger is activated, the event handler is called, which updates the state. The UI is automatically re-rendered to reflect the new state.

What is the

@rx.event

decorator?

Event handlers with arguments

Event handlers can also take in arguments. For example, the

increment

event handler can take an argument to increment the count by a specific amount.

0

Increment by 1

Increment by 5

The

on click

event trigger doesn't pass any arguments here, but some event triggers do. For example, the

on_blur

event trigger passes the text of an input as an argument to the event handler.

Make sure that the event handler has the same number of arguments as the event trigger, or an error will be raised.

Compile-time vs. runtime (IMPORTANT)

Before we dive deeper into state, it's important to understand the difference between compile-time and runtime in Reflex.

When you run your app, the frontend gets compiled to Javascript code that runs in the browser (compile-time). The backend stays in Python and runs on the server during the lifetime of the app (runtime).

When can you not use pure Python?

We cannot compile arbitrary Python code, only the components that you define. What this means importantly is that you cannot use arbitrary Python operations and functions on state vars in components.

However, since any event handlers in your state are on the backend, you

can use any Python code or library
within your state.
Examples that work
Within an event handler, use any Python code or library.
even
Increment
Use any Python function within components, as long as it is defined at compile time (i.e. does not reference
any state var)
0
true
1
false
2
true
3
false
4
true
5
false
6
true
7
false
8
true
9
false
Examples that don't work
You cannot do an
if
statement on vars in components, since the value is not known at compile time.
You cannot do a
for

loop over a list of vars.
You cannot do arbitrary Python operations on state vars in components.
In the next sections, we will show how to handle these cases.
Conditional rendering
As mentioned above, you cannot use Python
if/else
statements with state vars in components. Instead, use the
rx.cond
function to conditionally render components.
Not Logged In
Toggle Login
Rendering lists
To iterate over a var that is a list, use the
rx.foreach
function to render a list of components.
Pass the list var and a function that returns a component as arguments to
rx.foreach
•
Apple
Apple Banana
Banana
Banana Cherry
Banana Cherry The function that renders each item takes in a
Banana Cherry The function that renders each item takes in a Var
Banana Cherry The function that renders each item takes in a Var , since this will get compiled up front.
Banana Cherry The function that renders each item takes in a Var , since this will get compiled up front. Var Operations
Banana Cherry The function that renders each item takes in a Var , since this will get compiled up front. Var Operations You can't use arbitrary Python operations on state vars in components, but Reflex has
Banana Cherry The function that renders each item takes in a Var , since this will get compiled up front. Var Operations You can't use arbitrary Python operations on state vars in components, but Reflex has var operations
Banana Cherry The function that renders each item takes in a Var , since this will get compiled up front. Var Operations You can't use arbitrary Python operations on state vars in components, but Reflex has var operations that you can use to manipulate state vars.
Banana Cherry The function that renders each item takes in a Var , since this will get compiled up front. Var Operations You can't use arbitrary Python operations on state vars in components, but Reflex has var operations that you can use to manipulate state vars. For example, to check if a var is even, you can use the
Banana Cherry The function that renders each item takes in a Var , since this will get compiled up front. Var Operations You can't use arbitrary Python operations on state vars in components, but Reflex has var operations that you can use to manipulate state vars. For example, to check if a var is even, you can use the %
Banana Cherry The function that renders each item takes in a Var , since this will get compiled up front. Var Operations You can't use arbitrary Python operations on state vars in components, but Reflex has var operations that you can use to manipulate state vars. For example, to check if a var is even, you can use the % and

Even

Increment

App and Pages

Reflex apps are created by instantiating the

rx.App

class. Pages are linked to specific URL routes, and are created by defining a function that returns a component.

Next Steps

Now that you have a basic understanding of how Reflex works, the next step is to start coding your own apps. Try one of the following tutorials:

Dashboard Tutorial

Chatapp Tutorial

https://reflex.dev/docs/getting-started/chatapp-tutorial

Interactive Tutorial: Al Chat App

This tutorial will walk you through building an AI chat app with Reflex. This app is fairly complex, but don't worry - we'll break it down into small steps.

You can find the full source code for this app

here

What You'll Learn

In this tutorial you'll learn how to:

Install

reflex

and set up your development environment.

Create components to define and style your UI.

Use state to add interactivity to your app.

Deploy your app to share with others.

Setting up Your Project

Video: Example of Setting up the Chat App

We will start by creating a new project and setting up our development environment. First, create a new directory for your project and navigate to it.

Next, we will create a virtual environment for our project. This is optional, but recommended. In this example, we will use

venv

to create our virtual environment.

Now, we will install Reflex and create a new project. This will create a new directory structure in our project directory.

Note:

When prompted to select a template, choose option 0 for a blank project.

You can run the template app to make sure everything is working.

You should see your app running at

http://localhost:3000

.

Reflex also starts the backend server which handles all the state management and communication with the frontend. You can test the backend server is running by navigating to

http://localhost:8000/ping

.

Now that we have our project set up, in the next section we will start building our app!

Basic Frontend

Let's start with defining the frontend for our chat app. In Reflex, the frontend can be broken down into independent, reusable components. See the

components docs

for more information.

Display A Question And Answer

We will modify the

index

function in

chatapp/chatapp.py

file to return a component that displays a single question and answer.

What is Reflex?

A way to build web apps in pure Python!

Components can be nested inside each other to create complex layouts. Here we create a parent container that contains two boxes for the question and answer.

We also add some basic styling to the components. Components take in keyword arguments, called props

, that modify the appearance and functionality of the component. We use the

text_align

prop to align the text to the left and right.

Reusing Components

Now that we have a component that displays a single question and answer, we can reuse it to display multiple questions and answers. We will move the component to a separate function

question answer

and call it from the

index

function.

What is Reflex?

A way to build web apps in pure Python!

What can I make with it?

Anything from a simple website to a complex web app!

Chat Input Now we want a way for the user to input a question. For this, we will use the input component to have the user add text and a button component to submit the question. What is Reflex? A way to build web apps in pure Python! What can I make with it? Anything from a simple website to a complex web app! Ask Styling Let's add some styling to the app. More information on styling can be found in the styling docs . To keep our code clean, we will move the styling to a separate file chatapp/style.py We will import the styles in chatapp.py and use them in the components. At this point, the app should look like this: What is Reflex? A way to build web apps in pure Python! What can I make with it? Anything from a simple website to a complex web app! Ask The app is looking good, but it's not very useful yet! In the next section, we will add some functionality to the app. State Now let's make the chat app interactive by adding state. The state is where we define all the variables that can change in the app and all the functions that can modify them. You can learn more about state in the state docs **Defining State** We will create a new file called

state.py
in the
chatapp
directory. Our state will keep track of the current question being asked and the chat history. We will also
define an event handler
answer
which will process the current question and add the answer to the chat history.
Binding State to Components
Now we can import the state in
chatapp.py
and reference it in our frontend components. We will modify the
chat
component to use the state instead of the current fixed questions and answers.
Ask
Normal Python
for
loops don't work for iterating over state vars because these values can change and aren't known at compile
time. Instead, we use the
foreach
component to iterate over the chat history.
We also bind the input's
on_change
event to the
set_question
event handler, which will update the
question
state var while the user types in the input. We bind the button's
on_click
event to the
answer
event handler, which will process the question and add the answer to the chat history. The
set_question
event handler is a built-in implicitly defined event handler. Every base var has one. Learn more in the
events docs

under the Setters section. Clearing the Input Currently the input doesn't clear after the user clicks the button. We can fix this by binding the value of the input to question , with value=State.question , and clear it when we run the event handler for answer , with self.question = " Ask Streaming Text Normally state updates are sent to the frontend when an event handler returns. However, we want to stream the text from the chatbot as it is generated. We can do this by yielding from the event handler. See the yield events docs for more info. Ask In the next section, we will finish our chatbot by adding Al! Final App We will use OpenAl's API to give our chatbot some intelligence. Configure the OpenAl API Key First, ensure you have an active OpenAI subscription. Next, install the latest openai package: Direct Configuration of API in Code Update the state.py file to include your API key directly: Using the API Making your chatbot intelligent requires connecting to a language model API. This section explains how to integrate with OpenAI's API to power your chatbot's responses. First, the user types a prompt that is updated via the on_change event handler. Next, when a prompt is ready, the user can choose to submit it by clicking the

Ask
button which in turn triggers the
State.answer
method inside our
state.py
file.
Finally, if the method is triggered, the
orompt
s sent via a request to OpenAl client and returns an answer that we can trim and use to update the char
history!
Finally, we have our chatbot!
Final Code
This application is a simple, interactive chatbot built with Reflex that leverages OpenAI's API for intelligent
responses. The chatbot features a clean interface with streaming responses for a natural conversation
experience.
Key Features
Real-time streaming responses
Clean, visually distinct chat bubbles for questions and answers
Simple input interface with question field and submit button
Project Structure
Below is the full chatbot code with a commented title that corresponds to the filename.
The
chatapp.py
file:
The
state.py
file:
The
style.py
file:
Next Steps
Congratulations! You have built your first chatbot. From here, you can read through the rest of the
documentations to learn about Reflex in more detail. The best way to learn is to build something, so try to

build your own app using this as a starting point!

One More Thing

With our hosting service, you can deploy this app with a single command within minutes. Check out our Hosting Quick Start

.

https://reflex.dev/docs/getting-started/dashboard-tutorial

Tutorial: Data Dashboard

During this tutorial you will build a small data dashboard, where you can input data and it will be rendered in table and a graph. This tutorial does not assume any existing Reflex knowledge, but we do recommend checking out the quick

Basics Guide

first.

The techniques you'll learn in the tutorial are fundamental to building any Reflex app, and fully understanding it will give you a deep understanding of Reflex.

This tutorial is divided into several sections:

Setup for the Tutorial

: A starting point to follow the tutorial

Overview

: The fundamentals of Reflex UI (components and props)

Showing Dynamic Data

: How to use State to render data that will change in your app.

Add Data to your App

: Using a Form to let a user add data to your app and introduce event handlers.

Plotting Data in a Graph

: How to use Reflex's graphing components.

Final Cleanup and Conclusion

: How to further customize your app and add some extra styling to it.

What are you building?

In this tutorial, you are building an interactive data dashboard with Reflex.

You can see what the finished app and code will look like here:

Add User

Name

Email

Gender

Danilo Sousa

danilo@example.com

Male

Zahra Ambessa

Female Don't worry if you don't understand the code above, in this tutorial we are going to walk you through the whole thing step by step. Setup for the tutorial Check out the installation docs to get Reflex set up on your machine. Follow these to create a folder called dashboard_tutorial , which you will cd into and pip install reflex We will choose template when we run reflex init to get the blank template. Finally run reflex run to start the app and confirm everything is set up correctly. Overview Now that you're set up, let's get an overview of Reflex! Inspecting the starter code Within our dashboard_tutorial folder we just cd 'd into, there is a rxconfig.py file that contains the configuration for our Reflex app. (Check out the config docs for more information)

zahra@example.com

There is also an

assets

folder where static files such as images and stylesheets can be placed to be referenced within your app. (

asset docs

for more information)

Most importantly there is a folder also called

dashboard_tutorial

which contains all the code for your app. Inside of this folder there is a file named

dashboard_tutorial.py

. To begin this tutorial we will delete all the code in this file so that we can start from scratch and explain every step as we go.

The first thing we need to do is import

reflex

. Once we have done this we can create a component, which is a reusable piece of user interface code.

Components are used to render, manage, and update the UI elements in your application.

Let's look at the example below. Here we have a function called

index

that returns a

text

component (an in-built Reflex UI component) that displays the text "Hello World!".

Next we define our app using

app = rx.App()

and add the component we just defined (

index

) to a page using

app.add_page(index)

. The function name (in this example

index

) which defines the component, must be what we pass into the

add_page

. The definition of the app and adding a component to a page are required for every Reflex app.

This code will render a page with the text "Hello World!" when you run your app like below:

Hello World!

For the rest of the tutorial the

app=rx.App()

and
app.add_page
will be implied and not shown in the code snippets.
Creating a table
Let's create a new component that will render a table. We will use the
table
component to do this. The
table
component has a
root
, which takes in a
header
and a
body
, which in turn take in
row
components. The
row
component takes in
cell
components which are the actual data that will be displayed in the table.
Name
Email
Gender
Danilo Sousa
danilo@example.com
Male
Zahra Ambessa
zahra@example.com
Female
Components in Reflex have
props
, which can be used to customize the component and are passed in as keyword arguments to the component
function.

The
rx.table.root
component has for example the
variant
and
size
props, which customize the table as seen below.
Name
Email
Gender
Danilo Sousa
danilo@example.com
Male
Zahra Ambessa
zahra@example.com
Female
Showing dynamic data (State)
Up until this point all the data we are showing in the app is static. This is not very useful for a data dashboard.
We need to be able to show dynamic data that can be added to and updated.
This is where
State
comes in.
State
is a Python class that stores variables that can change when the app is running, as well as the functions that
can change those variables.
To define a state class, subclass
rx.State
and define fields that store the state of your app. The state variables (vars) should have a type annotation,
and can be initialized with a default value. Check out the
basics
section for a simple example of how state works.
In the example below we define a
State
class called

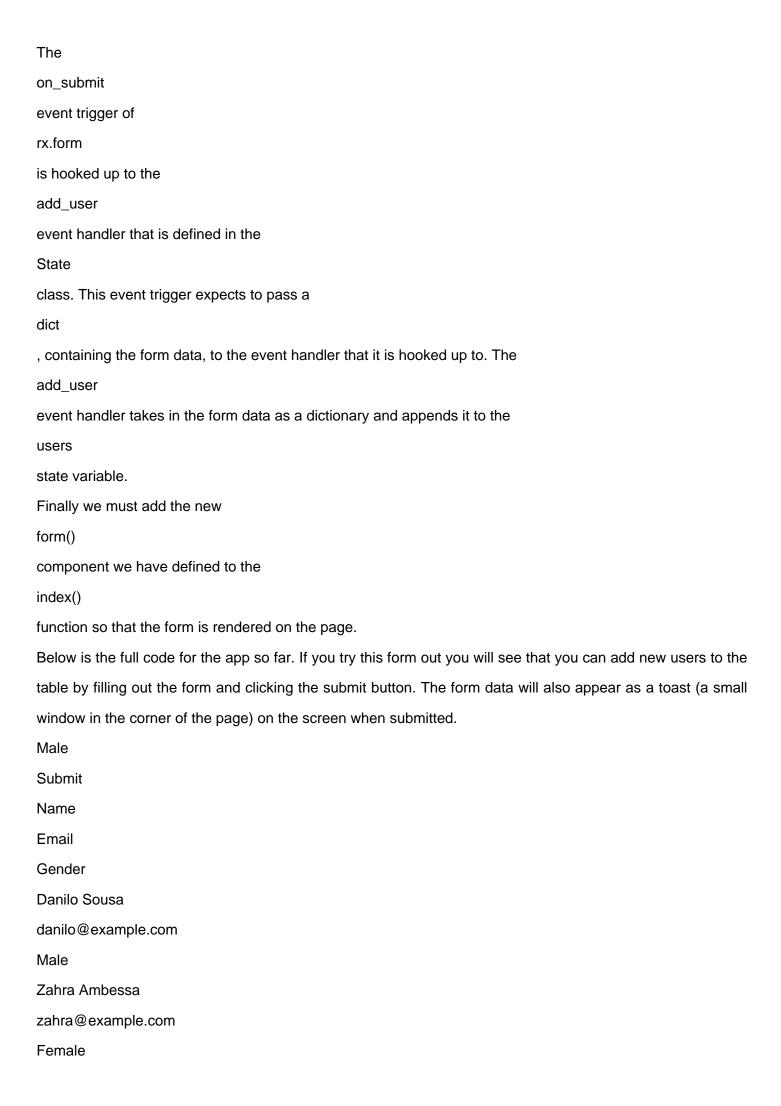
State
that has a variable called
users
that is a list of lists of strings. Each list in the
users
list represents a user and contains their name, email and gender.
To iterate over a state var that is a list, we use the
rx.foreach
function to render a list of components. The
rx.foreach
component takes an
iterable
(list, tuple or dict) and a
function
that renders each item in the
iterable
Why can we not just splat this in a
for
loop
Here the render function is
show_user
which takes in a single user and returns a
table.row
component that displays the users name, email and gender.
Name
Email
Gender
Danilo Sousa
danilo@example.com
Male
Zahra Ambessa
zahra@example.com
Female

As you can see the output above looks the same as before, except now the data is no longer static and can
change with user input to the app.
Using a proper class structure for our data
So far our data has been defined in a list of lists, where the data is accessed by index i.e.
user[0]
1
user[1]
. This is not very maintainable as our app gets bigger.
A better way to structure our data in Reflex is to use a class to represent a user. This way we can access the
data using attributes i.e.
user.name
,
user.email
In Reflex when we create these classes to showcase our data, the class must inherit from
rx.Base
•
rx.Base
is also necessary if we want to have a state var that is an iterable with different types. For example if we
wanted to have
age
as an
int
we would have to use
rx.base
as we could not do this with a state var defined as
list[list[str]]
The
show_user
render function is also updated to access the data by named attributes, instead of indexing.
Name
Email
Gender

Danilo Sousa
danilo@example.com
Male
Zahra Ambessa
zahra@example.com
Female
Next let's add a form to the app so we can add new users to the table.
Using a Form to Add Data
We build a form using
rx.form
, which takes several components such as
rx.input
and
rx.select
, which represent the form fields that allow you to add information to submit with the form. Check out the
form
docs for more information on form components.
The
rx.input
component takes in several props. The
placeholder
prop is the text that is displayed in the input field when it is empty. The
name
prop is the name of the input field, which gets passed through in the dictionary when the form is submitted.
The
required
prop is a boolean that determines if the input field is required.
The
rx.select
component takes in a list of options that are displayed in the dropdown. The other props used here are
identical to the
rx.input
component.
Male

This form is all very compact as you can see from the example, so we need to add some styling to make it look better. We can do this by adding a vstack component around the form fields. The vstack component stacks the form fields vertically. Check out the layout docs for more information on how to layout your app. Male Now you have probably realised that we have all the form fields, but we have no way to submit the form. We can add a submit button to the form by adding a rx.button component to the vstack component. The rx.button component takes in the text that is displayed on the button and the type prop which is the type of button. The type prop is set to submit so that the form is submitted when the button is clicked. In addition to this we need a way to update the users state variable when the form is submitted. All state changes are handled through functions in the state class, called event handlers Components have special props called event triggers, such as on_submit , that can be used to make components interactive. Event triggers connect components to event handlers, which update the state. Different event triggers expect the event handler that you hook them up to, to take in

different arguments (and some do not take in any arguments).



Putting the Form in an Overlay

In Reflex, we like to make the user interaction as intuitive as possible. Placing the form we just constructed in an overlay creates a focused interaction by dimming the background, and ensures a cleaner layout when you have multiple action points such as editing and deleting as well.

We will place the form inside of a

rx.dialog

component (also called a modal). The

rx.dialog.root

contains all the parts of a dialog, and the

rx.dialog.trigger

wraps the control that will open the dialog. In our case the trigger will be an

rx.button

that says "Add User" as shown below.

After the trigger we have the

rx.dialog.content

which contains everything within our dialog, including a title, a description and our form. The first way to close the dialog is without submitting the form and the second way is to close the dialog by submitting the form as shown below. This requires two

rx.dialog.close

components within the dialog.

The total code for the dialog with the form in it is below.

Add User

At this point we have an app that allows you to add users to a table by filling out a form. The form is placed in a dialog that can be opened by clicking the "Add User" button. We change the name of the component from form

to

add customer button

and update this in our

index

component. The full app so far and code are below.

Add User

Name

Email

Gender

Danilo Sousa danilo@example.com Male Zahra Ambessa zahra@example.com Female Plotting Data in a Graph The last part of this tutorial is to plot the user data in a graph. We will use Reflex's built-in graphing library recharts to plot the number of users of each gender. Transforming the data for the graph The graphing components in Reflex expect to take in a list of dictionaries. Each dictionary represents a data point on the graph and contains the x and y values. We will create a new event handler in the state called transform_data to transform the user data into the format that the graphing components expect. We must also create a new state variable called users_for_graph to store the transformed data, which will be used to render the graph. As we can see above the transform_data event handler uses the Counter class from the collections module to count the number of users of each gender. We then create a list of dictionaries from this which we set to the state var users_for_graph Finally we can see that whenever we add a new user through submitting the form and running the add_user event handler, we call the transform data event handler to update the users_for_graph state variable.

```
Rendering the graph
We use the
rx.recharts.bar chart
component to render the graph. We pass through the state variable for our graphing data as
data=State.users_for_graph
. We also pass in a
rx.recharts.bar
component which represents the bars on the graph. The
rx.recharts.bar
component takes in the
data_key
prop which is the key in the data dictionary that represents the y value of the bar. The
stroke
and
fill
props are used to set the color of the bars.
The
rx.recharts.bar_chart
component also takes in
rx.recharts.x_axis
and
rx.recharts.y_axis
components which represent the x and y axes of the graph. The
data_key
prop of the
rx.recharts.x_axis
component is set to the key in the data dictionary that represents the x value of the bar. Finally we add
width
and
height
props to set the size of the graph.
Finally we add this
graph()
component to our
```

index() component so that the graph is rendered on the page. The code for the full app with the graph included is below. If you try this out you will see that the graph updates whenever you add a new user to the table. Add User Name **Email** Gender Danilo Sousa danilo@example.com Male Zahra Ambessa zahra@example.com Female One thing you may have noticed about your app is that the graph does not appear initially when you run the app, and that you must add a user to the table for it to first appear. This occurs because the transform_data event handler is only called when a user is added to the table. In the next section we will explore a solution to this. Final Cleanup Revisiting app.add_page At the beginning of this tutorial we mentioned that the app.add_page function is required for every Reflex app. This function is used to add a component to a page. The app.add_page currently looks like this app.add_page(index) . We could change the route that the page renders on by setting the route prop such as route="/custom-route"

, this would change the route to

for this page.

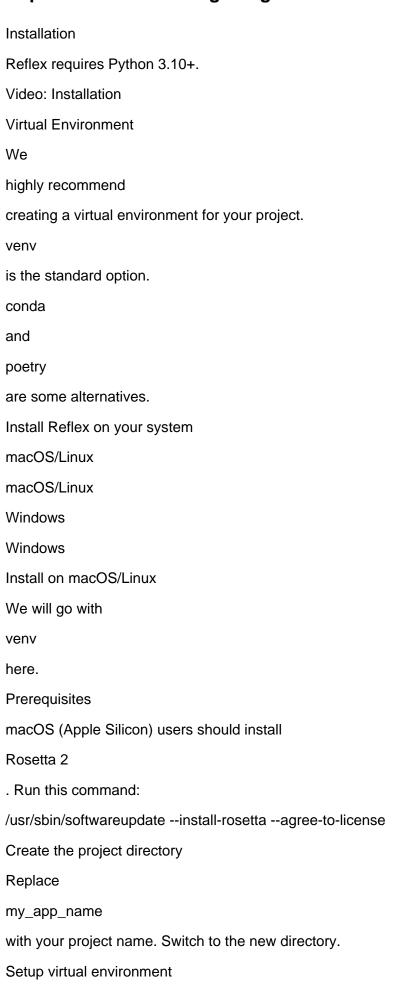
http://localhost:3000/custom-route

We can also set a
title
to be shown in the browser tab and a
description
as shown in search results.
To solve the problem we had above about our graph not loading when the page loads, we can use
on_load
inside of
app.add_page
to call the
transform_data
event handler when the page loads. This would look like
on_load=State.transform_data
. Below see what our
app.add_page
would look like with some of the changes above added.
Add User
Name
Email
Gender
Danilo Sousa
danilo@example.com
Male
Zahra Ambessa
zahra@example.com
Female
Revisiting app=rx.App()
At the beginning of the tutorial we also mentioned that we defined our app using
app=rx.App()
. We can also pass in some props to the
гх.Арр
component to customize the app.
The most important one is
theme

which allows you to customize the look and feel of the app. The theme prop takes in an rx.theme component which has several props that can be set. The radius prop sets the global radius value for the app that is inherited by all components that have a radius prop. It can be overwritten locally for a specific component by manually setting the radius prop. The accent_color prop sets the accent color of the app. Check out other options for the accent color here To see other props that can be set at the app level check out this documentation Unfortunately in this tutorial here we cannot actually apply this to the live example on the page, but if you copy and paste the code below into a reflex app locally you can see it in action. Conclusion Finally let's make some final styling updates to our app. We will add some hover styling to the table rows and center the table inside the show_user with style=\{"_hover": {"bg": rx.color("gray", 3)}}, align="center" In addition, we will add some width="100%" and align="center" to the index()

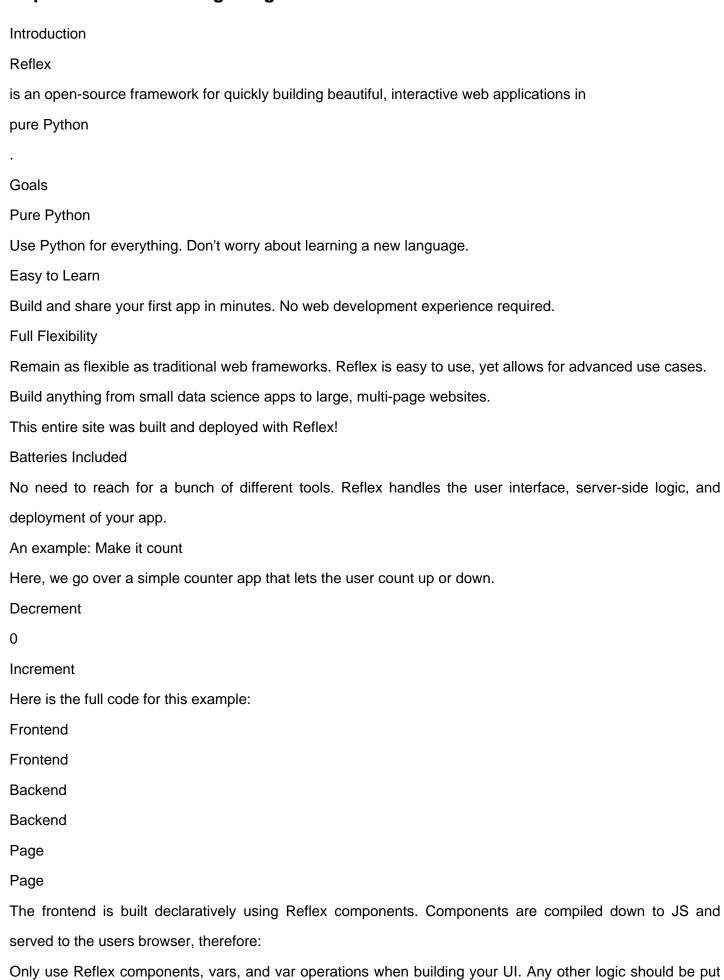
component to center the items on the page and ensure they stretch the full width of the page. Check out the full code and interactive app below: Add User Name **Email** Gender Danilo Sousa danilo@example.com Male Zahra Ambessa zahra@example.com Female And that is it for your first dashboard tutorial. In this tutorial we have created a table to display user data a form to add new users to the table a dialog to showcase the form a graph to visualize the user data In addition to the above we have we have explored state to allow you to show dynamic data that changes over time explored events to allow you to make your app interactive and respond to user actions added styling to the app to make it look better Advanced Section (Hooking this up to a Database) Coming Soon!

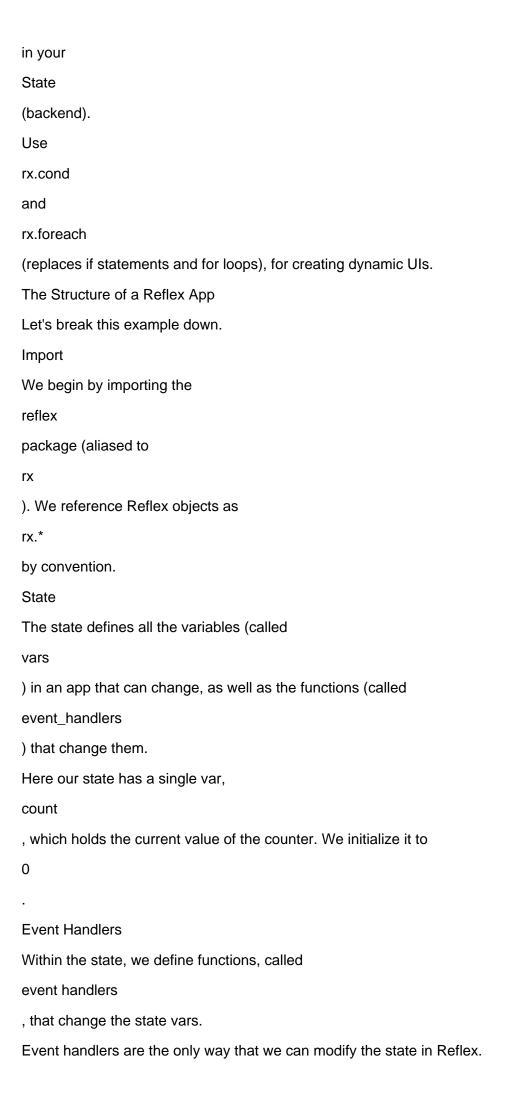
https://reflex.dev/docs/getting-started/installation



Getting
No module named venv
?
Install Reflex package
Reflex is available as a
pip
package.
Getting
command not found: pip
?
Initialize the project
Error
command not found: reflex
Mac / Linux
The command will return four template options to choose from as shown below.
From here select a template.
Run the App
Run it in development mode:
Your app runs at
http://localhost:3000
Reflex prints logs to the terminal. To increase log verbosity to help with debugging, use the
loglevel
flag:
Reflex will
hot reload
any code changes in real time when running in development mode. Your code edits will show up on
http://localhost:3000
automatically.

https://reflex.dev/docs/getting-started/introduction



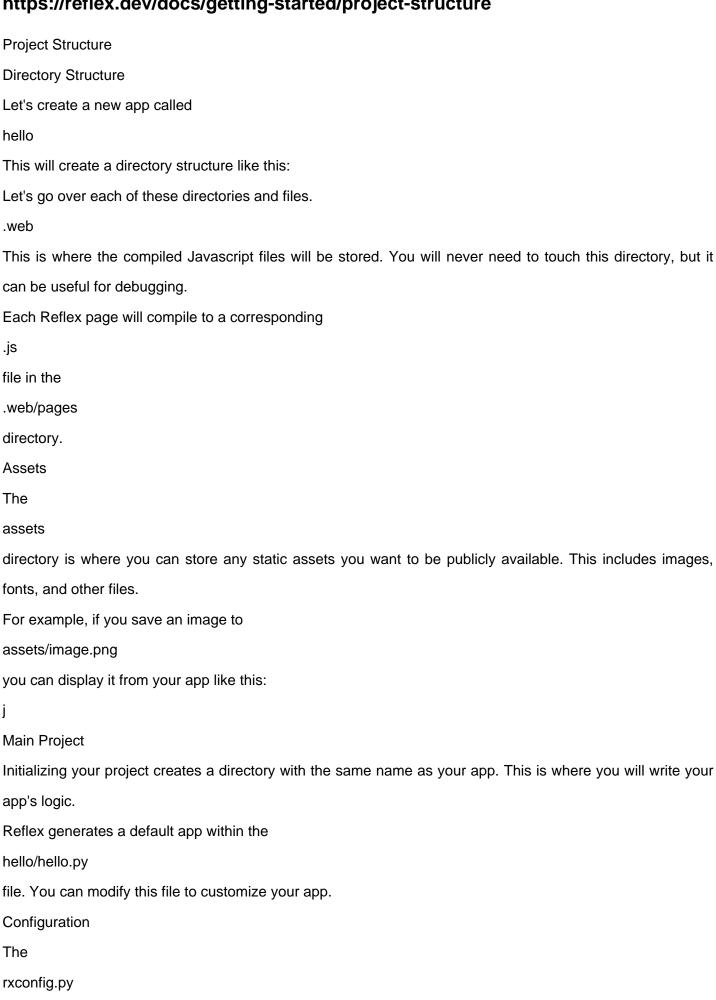


They can be called in response to user actions, such as clicking a button or typing in a text box.
These actions are called
events
Our counter app has two event handlers,
increment
and
decrement
User Interface (UI)
This function defines the app's user interface.
We use different components such as
rx.hstack
,
rx.button
, and
rx.heading
to build the frontend. Components can be nested to create complex layouts, and can be styled using the full
power of CSS.
Reflex comes with
50+ built-in components
to help you get started.
We are actively adding more components. Also, it's easy to
wrap your own React components
•
Components can reference the app's state vars.
The
rx.heading
component displays the current value of the counter by referencing
State.count State.count
All components that reference state will reactively update whenever the state changes.
Components interact with the state by binding events triggers to event handlers.
For example,

on_click is an event that is triggered when a user clicks a component. The first button in our app binds its on_click event to the State.decrement event handler. Similarly the second button binds on_click to State.increment In other words, the sequence goes like this: User clicks "increment" on the UI. on_click event is triggered. Event handler State.increment is called. State.count is incremented. UI updates to reflect the new value of State.count Add pages Next we define our app and add the counter component to the base route. **Next Steps** 🎉 And that's it! We've created a simple, yet fully interactive web app in pure Python. By continuing with our documentation, you will learn how to building awesome apps with Reflex. For a glimpse of the possibilities, check out these resources: For a more real-world example, check out either the dashboard tutorial or the chatapp tutorial

We have bots that can answer questions and generate Reflex code for you. Check them out in #ask-ai in our Discord

https://reflex.dev/docs/getting-started/project-structure



file can be used to configure your app. By default it looks something like this: We will discuss project structure and configuration in more detail in the advanced project structure documentation.

https://reflex.dev/docs/hosting/adding-members

Project

A project is a collection of applications (apps / websites).

Every project has its own billing page that are accessible to Admins.

Adding Team Members

To see the team members of a project click on the

Members

tab in the Cloud UI on the project page.

If you are a User you have the ability to create, deploy and delete apps, but you do not have the power to add or delete users from that project. You must be an Admin for that.

As an Admin you will see the an

Add user

button in the top right of the screen, as shown in the image below. Clicking on this will allow you to add a user to the project. You will need to enter the email address of the user you wish to add.

Currently a User must already have logged in once before they can be added to a project.

Other project settings

Clicking on the

Settings

tab in the Cloud UI on the project page allows a user to change the

project name

, check the

project id

and, if the project is not your default project, delete the project.

https://reflex.dev/docs/hosting/app-management

App

In Reflex Cloud an "app" (or "application" or "website") refers to a web application built using the Reflex framework, which can be deployed and managed within the Cloud platform.

You can deploy an app using the

reflex deploy

command.

There are many actions you can take in the Cloud UI to manage your app. Below are some of the most common actions you may want to take.

Stopping an App

To stop an app follow the arrow in the image below and press on the

Stop app

button. Pausing an app will stop it from running and will not be accessible to users until you resume it. In addition, this will stop you being billed for your app.

CLI Command to stop an app

Deleting an App

To delete an app click on the

Settings

tab in the Cloud UI on the app page.

Then click on the

Danger

tab as shown below.

Here there is a

Delete app

button. Pressing this button will delete the app and all of its data. This action is irreversible.

CLI Command to delete an app

Other app settings

Clicking on the

Settings

tab in the Cloud UI on the app page also allows a user to change the

app name

, change the

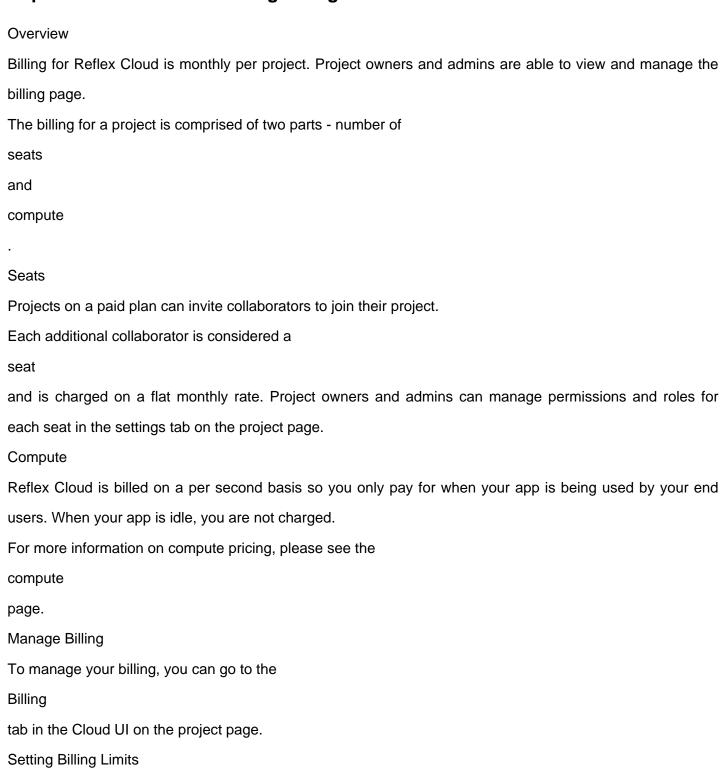
app description

app id
.
The other app settings also allows users to edit and add secrets (environment variables) to the app. For more information on secrets, see the
Secrets (Environment Variables)

and check the

page.

https://reflex.dev/docs/hosting/billing



If you want to set a billing limit for your project, you can do so by going to the

Billing

tab in the Cloud UI on the project page.

https://reflex.dev/docs/hosting/cli/apps

reflex cloud apps scale Scale an application by changing the VM type or adding/removing regions. Usage **Options** --app-name TEXT : The name of the app. --vm-type TEXT : The virtual machine type to scale to. --regions, -r TEXT : Region to scale the app to. --token TEXT : The authentication token. --loglevel [debug|default|info|warning|error|critical] : The log level to use. --scale-type TEXT : The type of scaling. --interactive, -i / --no-interactive : Whether to use interactive mode. --help : Show this message and exit. reflex cloud apps status Retrieve the status of a specific deployment. Usage **Options** --watch / --no-watch : Whether to continuously watch the status. --token TEXT : The authentication token. --loglevel [debug|default|info|warning|error|critical] : The log level to use. --interactive, -i / --no-interactive

: Whether to use interactive mode.

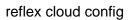
help
: Show this message and exit.
reflex cloud apps start
Start a stopped application.
Usage
Options
app-name TEXT
: The name of the application.
token TEXT
: The authentication token.
loglevel [debug default info warning error critical]
: The log level to use.
interactive, -i /no-interactive
: Whether to use interactive mode.
help
: Show this message and exit.
reflex cloud apps stop
Stop a running application.
Stop a running application. Usage
Usage
Usage Options
Usage Optionsapp-name TEXT
Usage Optionsapp-name TEXT : The name of the application.
Usage Optionsapp-name TEXT : The name of the applicationtoken TEXT
Usage Optionsapp-name TEXT : The name of the applicationtoken TEXT : The authentication token.
Usage Optionsapp-name TEXT : The name of the applicationtoken TEXT : The authentication tokenloglevel [debug default info warning error critical]
Usage Optionsapp-name TEXT : The name of the applicationtoken TEXT : The authentication tokenloglevel [debug default info warning error critical] : The log level to use.
Usage Optionsapp-name TEXT : The name of the applicationtoken TEXT : The authentication tokenloglevel [debug default info warning error critical] : The log level to useinteractive, -i /no-interactive
Usage Optionsapp-name TEXT : The name of the applicationtoken TEXT : The authentication tokenloglevel [debug default info warning error critical] : The log level to useinteractive, -i /no-interactive : Whether to use interactive mode.
Usage Optionsapp-name TEXT : The name of the applicationtoken TEXT : The authentication tokenloglevel [debug default info warning error critical] : The log level to useinteractive, -i /no-interactive : Whether to use interactive modehelp
Usage Optionsapp-name TEXT : The name of the applicationtoken TEXT : The authentication tokenloglevel [debug default info warning error critical] : The log level to useinteractive, -i /no-interactive : Whether to use interactive modehelp : Show this message and exit.
Usage Optionsapp-name TEXT : The name of the applicationtoken TEXT : The authentication tokenloglevel [debug default info warning error critical] : The log level to useinteractive, -i /no-interactive : Whether to use interactive modehelp : Show this message and exit. reflex cloud apps delete

app-name TEXT
: The name of the application.
token TEXT
: The authentication token.
loglevel [debug default info warning error critical]
: The log level to use.
interactive, -i /no-interactive
: Whether to use interactive mode.
help
: Show this message and exit.
reflex cloud apps logs
Retrieve logs for a given application.
Usage
Options
app-name TEXT
: The name of the application.
token TEXT
: The authentication token.
offset INTEGER
: The offset in seconds from the current time.
start INTEGER
: The start time in Unix epoch format.
end INTEGER
: The end time in Unix epoch format.
loglevel [debug default info warning error critical]
: The log level to use.
interactive, -i /no-interactive
: Whether to use interactive mode.
cursor TEXT
: The cursor for pagination.
pretty
: Use pretty printing for logs.
follow
: Asks to continue to query logs.

help
: Show this message and exit.
reflex cloud apps history
Retrieve the deployment history for a given application.
Usage
Options
app-name TEXT
: The name of the application.
token TEXT
: The authentication token.
loglevel [debug default info warning error critical]
: The log level to use.
json, -j /no-json
: Whether to output the result in json format.
interactive, -i /no-interactive
: Whether to use interactive mode.
help
: Show this message and exit.
reflex cloud apps build-logs
Retrieve the build logs for a specific deployment.
Usage
Options
token TEXT
: The authentication token.
interactive, -i /no-interactive
: Whether to use interactive mode.
help
: Show this message and exit.
reflex cloud apps list
List all the hosted deployments of the authenticated user. Will exit if unable to list deployments.
Usage
Options
project TEXT
: The project ID to filter deployments.

- --project-name TEXT
- : The name of the project.
- --token TEXT
- : The authentication token.
- --loglevel [debug|default|info|warning|error|critical]
- : The log level to use.
- --json, -j / --no-json
- : Whether to output the result in JSON format.
- --interactive / --no-interactive
- : Whether to list configuration options and ask for confirmation.
- --help
- : Show this message and exit.

https://reflex.dev/docs/hosting/cli/config



Generate a configuration file for the cloud deployment.

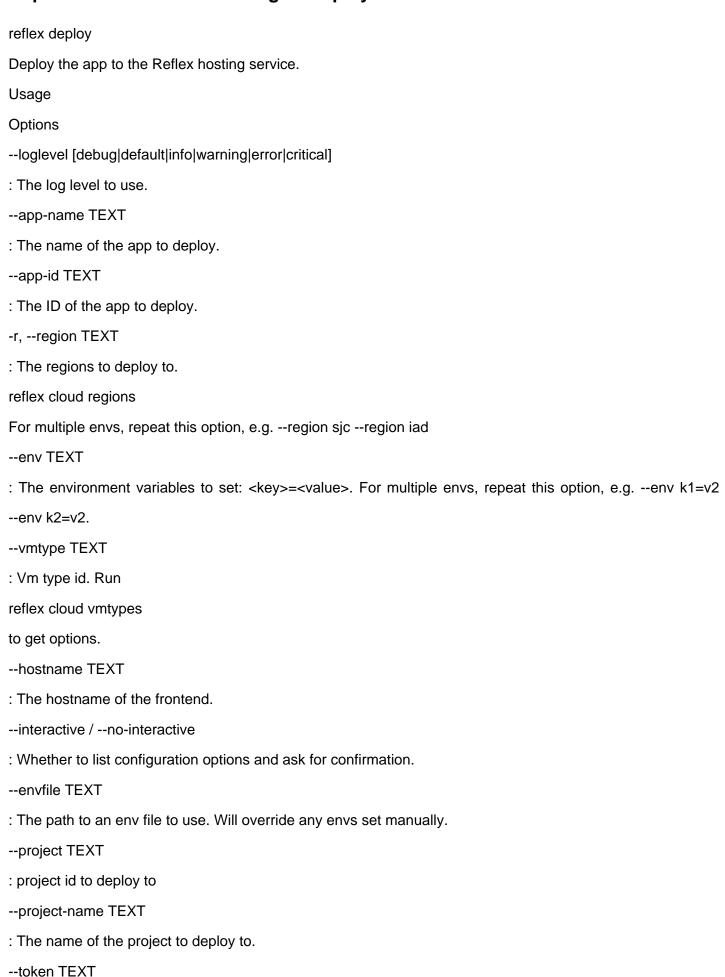
Usage

Options

--help

: Show this message and exit.

https://reflex.dev/docs/hosting/cli/deploy



- : token to use for auth
- --config-path, --config TEXT
- : path to the config file
- --help
- : Show this message and exit.

https://reflex.dev/docs/hosting/cli/login

reflex login Authenticate with experimental Reflex hosting service. Usage Options --loglevel [debug|default|info|warning|error|critical] : The log level to use. --help : Show this message and exit. reflex logout Log out of access to Reflex hosting service. Usage Options

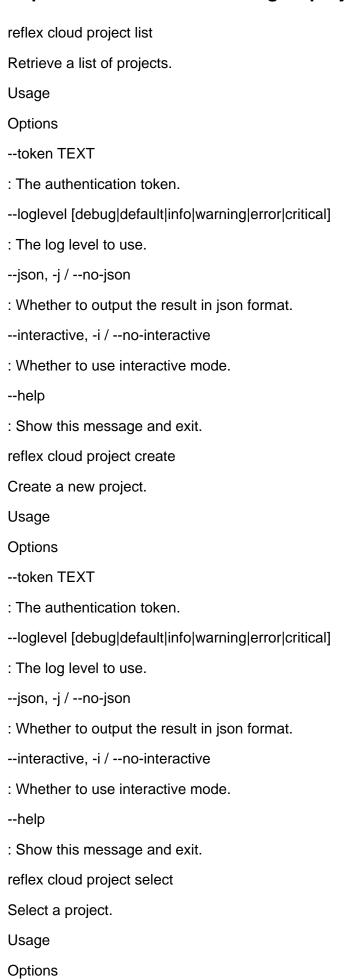
--loglevel [debug|default|info|warning|error|critical]

: The log level to use.

: Show this message and exit.

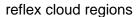
--help

https://reflex.dev/docs/hosting/cli/projects



: The log level to use. --json, -j / --no-json : Whether to output the result in json format. --interactive / --no-interactive : Whether to list configuration options and ask for confirmation. --help : Show this message and exit. reflex cloud project users Retrieve the users for a project. Usage **Options** --project-id TEXT : The ID of the project. If not provided, the selected project will be used. If no project is selected, it throws an error. --project-name TEXT : The name of the project. --token TEXT : The authentication token. --loglevel [debug|default|info|warning|error|critical] : The log level to use. --json, -j / --no-json : Whether to output the result in json format. --interactive / --no-interactive : Whether to list configuration options and ask for confirmation. --help : Show this message and exit.

https://reflex.dev/docs/hosting/cli/regions



List all the regions of the hosting service.

Areas available for deployment are:

ams Amsterdam, Netherlands

arn Stockholm, Sweden

atl Atlanta, Georgia (US)

bog BogotÃi, Colombia

bom Mumbai, India

bos Boston, Massachusetts (US)

cdg Paris, France

den Denver, Colorado (US)

dfw Dallas, Texas (US)

ewr Secaucus, NJ (US)

eze Ezeiza, Argentina

fra Frankfurt, Germany

gdl Guadalajara, Mexico

gig Rio de Janeiro, Brazil

gru Sao Paulo, Brazil

hkg Hong Kong, Hong Kong

iad Ashburn, Virginia (US)

jnb Johannesburg, South Africa

lax Los Angeles, California (US)

Ihr London, United Kingdom

mad Madrid, Spain

mia Miami, Florida (US)

nrt Tokyo, Japan

ord Chicago, Illinois (US)

otp Bucharest, Romania

phx Phoenix, Arizona (US)

qro Querétaro, Mexico

scl Santiago, Chile

sea Seattle, Washington (US)

sin Singapore, Singapore

sjc San Jose, California (US)

syd Sydney, Australia

waw Warsaw, Poland

yul Montreal, Canada

yyz Toronto, Canada.

Usage

Options

--loglevel [debug|default|info|warning|error|critical]

: The log level to use.

--json, -j / --no-json

: Whether to output the result in json format.

--help

: Show this message and exit.

https://reflex.dev/docs/hosting/cli/secrets

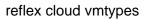
reflex cloud secrets list Retrieve secrets for a given application. Usage **Options** --token TEXT : The authentication token. --loglevel [debug|default|info|warning|error|critical] : The log level to use. --json, -j / --no-json : Whether to output the result in JSON format. --interactive, -i / --no-interactive : Whether to use interactive mode. --help : Show this message and exit. reflex cloud secrets delete Delete a secret for a given application. Usage **Options** --token TEXT : The authentication token. --reboot / --no-reboot : Automatically reboot your site with the new secrets --loglevel [debug|default|info|warning|error|critical] : The log level to use. --interactive, -i / --no-interactive : Whether to use interactive mode. --help : Show this message and exit. reflex cloud secrets update Update secrets for a given application.

Usage

Options

- --envfile TEXT
- : The path to an env file to use. Will override any envs set manually.
- --env TEXT
- : The environment variables to set: <key>=<value>. Required if envfile is not specified. For multiple envs, repeat this option, e.g. --env k1=v2 --env k2=v2.
- --reboot / --no-reboot
- : Automatically reboot your site with the new secrets
- --token TEXT
- : The authentication token.
- --loglevel [debug|default|info|warning|error|critical]
- : The log level to use.
- --interactive, -i / --no-interactive
- : Whether to use interactive mode.
- --help
- : Show this message and exit.

https://reflex.dev/docs/hosting/cli/vmtypes



Retrieve the available VM types.

Usage

Options

- --token TEXT
- : The authentication token.
- --loglevel [debug|default|info|warning|error|critical]
- : The log level to use.
- --json, -j / --no-json
- : Whether to output the result in json format.
- --help
- : Show this message and exit.

https://reflex.dev/docs/hosting/compute

Compute Usage

Reflex Cloud is billed on a per second basis so you only pay for when your app is being used by your end users. When your app is idle, you are not charged.

This allows you to deploy your app on larger sizes and multiple regions without worrying about paying for idle compute. We bill on a per second basis so you only pay for the compute you use.

By default your app stays alive for 5 minutes after the no users are connected. After this time your app will be considered idle and you will not be charged. Start up times usually take less than 1 second for you apps to come back online.

Warm vs Cold Start

Apps below

c2m2

are considered warm starts and are usually less than 1 second.

If your app is larger than

c2m2

it will be a cold start which takes around 15 seconds. If you want to avoid this you can reserve a machine.

Compute Pricing Table

Machine Sizes

Per min

Per hour

Machine

vCPU

GB RAM

Cost / min

c1m.5

1

0.5

\$0.000767

c1m1

1

1

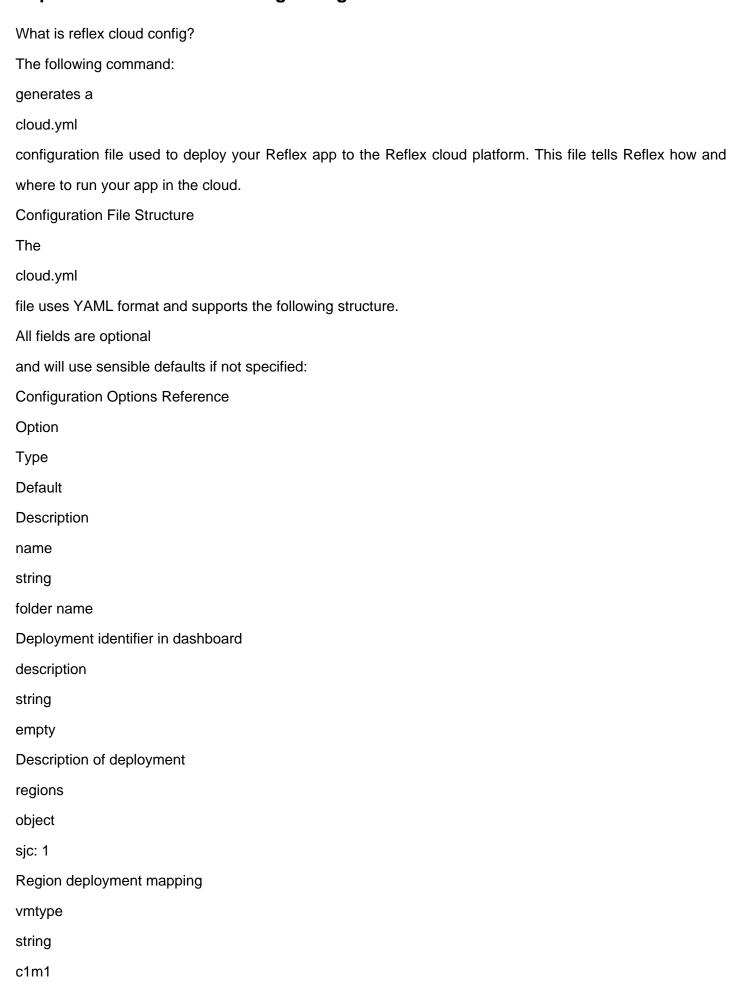
\$0.001383

c1m2

```
1
2
$0.002617
c2m2
2
2
$0.002767
c2m4
2
4
$0.005200
c4m4
4
4
$0.005533
c4m8
8
$0.010417
Reserved Machines (Coming Soon)
If you expect your apps to be continuously receiving users, you may want to reserve a machine instead of
having us manage your compute.
This will be a flat monthly rate for the machine.
Monitoring Usage
To monitor your projects usage, you can go to the billing tab in the Reflex Cloud UI on the project page.
Here you can see the current billing and usage for your project.
Real Life Examples of compute charges on the Pro tier
Single Application - Single Region
Single Application - Multi Region
Single Growing Application - Multi Region
Single Application High-Performance App - Single Region
Single Fast Scaling App - Multiple Regions
Multiple Apps - Multiple Regions
One thing that is important to note is that in the hypothetical example where you have
```

```
50 people
using your app
continuously for 24 hours
or if you have
1 person
using your app
continuously for 24 hours
, you
will be charged the same amount
as the charge is based on the amount of time your app up and not the number of users using your app. In
both these examples
your
app is up for 24 hours
and therefore you will be
charged for 24 hours of compute
```

https://reflex.dev/docs/hosting/config-file



Virtual machine specifications
hostname
string
null
Custom subdomain
envfile
string
.env
Environment variables file path
project
uuid
null
Project uuid
projectname
string
null
Project name
packages
array
empty
Additional system packages
include_db
boolean
false
Include local sqlite
strategy
string
auto
Deployment strategy
Configuration Options
For details of specific sections click the links in the table.
Projects
Organize deployments using projects:
You can also specify a project uuid instead of name:

You can go to the homepage of the project in the reflex cloud dashboard to find your project uuid in the url https://cloud.reflex.dev/project/uuid **Apt Packages** Install additional system packages your application requires. Package names are based on the apt package manager: Include SQLite Include local sqlite database: This is not persistent and will be lost on restart. It is recommended to use a database service instead. Strategy Deployment strategy: Available strategies: immediate : [Default] Deploy immediately rolling : Deploy in a rolling manner bluegreen : Deploy in a blue-green manner canary : Deploy in a canary manner, boot as single machine verify its health and then restart the rest. Multi-Environment Setup Development (cloud-dev.yml): Staging (cloud-staging.yml): Production (cloud-prod.yml): Deploy with specific configuration files:

https://reflex.dev/docs/hosting/custom-domains

Custom Domains

With the Pro tier of Reflex Cloud you can use your own custom domain to host your app.

Prerequisites

You must purchase a domain from a domain registrar such as GoDaddy, Cloudflare, Namecheap, or AWS.

For this tutorial we will use GoDaddy and the example domain

tomgotsman.us

.

Steps

Once you have purchased your domain, you can add it to your Reflex Cloud app by following these steps:

- 1 Ensure you have deployed your app to Reflex Cloud.
- 2 Once your app is deployed click the

Custom Domain

tab and add your custom domain to the input field and press the Add domain button. You should now see a page like below:

- 3 On the domain registrar's website, navigate to the DNS settings for your domain. It should look something like the image below:
- 4 Add all four of the DNS records provided by Reflex Cloud to your domain registrar's DNS settings. If there is already an A name record, delete it and replace it with the one provided by Reflex Cloud. Your DNS settings should look like the image below:

It may alert you that this record will resolve on #####.tomgotsman.us.tomgotsman.us.

Your domain provider may not support an Apex CNAME record, in this case just use an A record.

- 5 Once you have added the DNS records, refresh the page on the Reflex Cloud page (it may take a few minutes to a few hours to update successfully). If the records are correct, you should see a success message like the one below:
- 6 Now redeploy your app using the

reflex deploy

command and your app should now be live on your custom domain!

https://reflex.dev/docs/hosting/databricks

DATABRICKS_CATALOG

Deploy Reflex to Databricks This guide walks you through deploying a Reflex web application on Databricks using the Apps platform. Prerequisites Databricks workspace with Unity Catalog enabled GitHub repository containing your Reflex application Reflex Enterprise license (for single-port deployment) Step 1: Connect Your Repository Link GitHub Repository Navigate to your Databricks workspace Go to your user directory Click Create â†' Git folder Paste the URL of your GitHub repository containing the Reflex application Step 2: Configure Application Settings Create Configuration File Create a new file called app.yaml directly in Databricks (not in GitHub): **Obtain Required Tokens** Reflex Access Token Visit Reflex Cloud Tokens Navigate to Account Settings â†' Tokens Create a new token and copy the value Replace your-token-here in the configuration **Databricks Resources** Update

with your target catalog name
Update
DATABRICKS_SCHEMA
with your target schema name
Step 3: Enable Single-Port Deployment
Update your Reflex application for Databricks compatibility:
Update rxconfig.py
Update Application Entry Point
Modify your main application file where you define
rx.App
:
Also add
reflex-enterprise
and
asgiproxy
to your
requirements.txt
file.
Step 4: Create Databricks App
Navigate to Apps
Go to
Compute
→
Apps
Click
Create App
Configure Application
Select
Custom App
Configure SQL warehouse for your application
Step 5: Set Permissions
Catalog Permissions
Navigate to
Catalog

6±1. Coloot voi in torget estalor
→ Select your target catalog
Go to
Permissions
Add the app's service principal user
Grant the following permissions:
USE CATALOG
USE SCHEMA
Schema Permissions
Navigate to the specific schema
Go to
Permissions
Grant the following permissions:
USE SCHEMA
EXECUTE
SELECT
READ VOLUME
(if required)
Step 6: Deploy Application
Initiate Deployment
Click
Deploy
n the Apps interface
When prompted for the code path, provide your Git folder path or select your repository folder
Monitor Deployment
The deployment process will begin automatically
Monitor logs for any configuration issues
Updating Your Application
To deploy updates from your GitHub repository:
Pull Latest Changes
In the deployment interface, click
Deployment Source
Select
main
oranch

Click
Pull
to fetch the latest changes from GitHub
Redeploy
Click
Deploy
again to apply the updates
Configuration Reference
Environment Variable
Description
Example
HOME
Application home directory
/tmp/reflex
REFLEX_ACCESS_TOKEN
Authentication for Reflex Cloud
rx_token
DATABRICKS_WAREHOUSE_ID
SQL warehouse identifier
Auto-assigned
DATABRICKS_CATALOG
Target catalog name
main
DATABRICKS_SCHEMA
Target schema name
default
REFLEX_SHOW_BUILT_WITH_REFLEX
Show Reflex branding (Enterprise only)
0
or
1
Troubleshooting
Permission Errors
: Verify that all catalog and schema permissions are correctly set

P	∩r t	Iss	IIES
	וועו	155	いたう

: Ensure you're using

\$DATABRICKS_APP_PORT

and single-port configuration

Token Issues

: Verify your Reflex access token is valid and properly configured

Deployment Failures

: Check the deployment logs for specific error messages

Notes

Single-port deployment requires Reflex Enterprise

Configuration must be created directly in Databricks, not pushed from GitHub

Updates require manual pulling from the deployment interface

https://reflex.dev/docs/hosting/deploy-quick-start

Reflex Cloud - Quick Start

So far, we have been running our apps locally on our own machines.

But what if we want to share our apps with the world? This is where

the hosting service comes in.

Quick Start

Reflex's hosting service makes it easy to deploy your apps without worrying about configuring the infrastructure.

Prerequisites

Hosting service requires

reflex>=0.6.6

This tutorial assumes you have successfully

reflex init

and

reflex run

your app.

Also make sure you have a

requirements.txt

file at the top level app directory that contains all your python dependencies! (To create a

requirements.txt

file, run

pip freeze > requirements.txt

.)

Authentication

First run the command below to login / signup to your Reflex Cloud account: (command line)

You will be redirected to your browser where you can authenticate through Github or Gmail.

Web UI

Once you are at this URL and you have successfully authenticated, click on the one project you have in your workspace. You should get a screen like this:

This screen shows the login command and the deploy command. As we are already logged in, we can skip the login command.

Deployment

Now you can start deploying your app.

In your cloud UI copy the

reflex deploy

command similar to the one shown below.

In your project directory (where you would normally run

reflex run

) paste this command.

The command is by default interactive. It asks you a few questions for information required for the deployment.

The first question will compare your

requirements.txt

to your python environment and if they are different then it will ask you if you want to update your

requirements.txt

or to continue with the current one. If they are identical this question will not appear. To create a

requirements.txt

file. run

pip freeze > requirements.txt

The second question will search for a deployed app with the name of your current app, if it does not find one then it will ask if you wish to proceed in deploying your new app.

The third question is optional and will ask you for an app description.

That's it! You should receive some feedback on the progress of your deployment and in a few minutes your app should be up. 🎉

For detailed information about the deploy command and its options, see the

Deploy API Reference

and the

CLI Reference

Once your code is uploaded, the hosting service will start the deployment. After a complete upload, exiting from the command

does not

affect the deployment process. The command prints a message when you can safely close it without affecting the deployment.

If you go back to the Cloud UI you should be able to see your deployed app and other useful app information.

Setup a Cloud Config File

Moving around the Cloud UI

All flag values are saved between runs

https://reflex.dev/docs/hosting/deploy-with-github-actions

Deploy with Github Actions

This GitHub Action simplifies the deployment of Reflex applications to Reflex Cloud. It handles setting up the environment, installing the Reflex CLI, and deploying your app with minimal configuration.

This action requires

reflex>=0.6.6

Features:

Deploy Reflex apps directly from your GitHub repository to Reflex Cloud.

Supports subdirectory-based app structures.

Securely uses authentication tokens via GitHub Secrets.

Usage

Add the Action to Your Workflow

Create a

.github/workflows/deploy.yml

file in your repository and add the following:

Set Up Your Secrets

Store your Reflex authentication token securely in your repository's secrets:

Go to your GitHub repository.

Navigate to Settings > Secrets and variables > Actions > New repository secret.

Create new secrets for

REFLEX_AUTH_TOKEN

and

REFLEX_PROJECT_ID

(Create a

REFLEX_AUTH_TOKEN

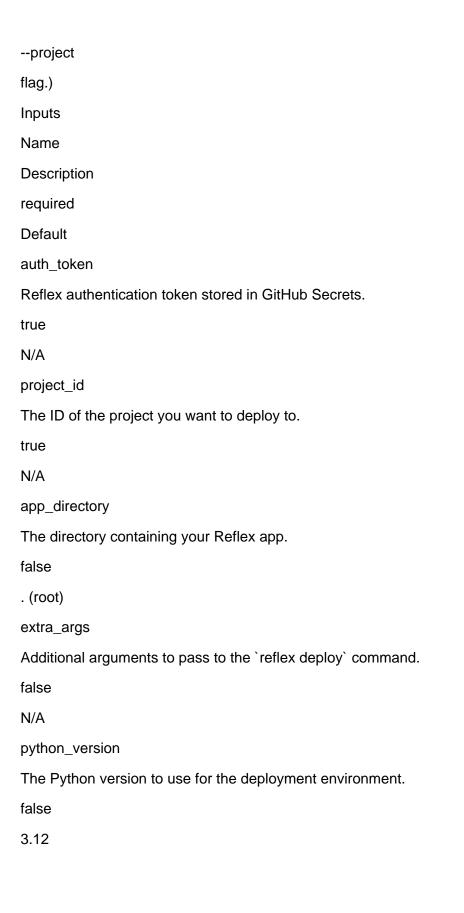
in the tokens tab of your UI, check out these

docs

The

REFLEX_PROJECT_ID

can be found in the UI when you click on the How to deploy button on the top right when inside a project and copy the ID after the



https://reflex.dev/docs/hosting/logs

View Logs
To view the app logs follow the arrow in the image below and press on the

dropdown.

Logs

CLI Command to view logs

View Deployment Logs and Deployment History

To view the deployment history follow the arrow in the image below and press on the

Deployments

.

This brings you to the page below where you can see the deployment history of your app. Click on deployment you wish to explore further.

CLI Command to view deployment history

This brings you to the page below where you can view the deployment logs of your app by clicking the

Build logs

dropdown.

https://reflex.dev/docs/hosting/machine-types

Machine Types

To scale your app you can choose different VMTypes. VMTypes are different configurations of CPU and RAM.

To scale your VM in the Cloud UI, click on the

Settings

tab in the Cloud UI on the app page, and then click on the

Scale

tab as shown below. Clicking on the

Change VM

button will allow you to scale your app.

VMTypes in the CLI

To get all the possible VMTypes you can run the following command:

To set which VMType to use when deploying your app you can pass the

--vmtype

flag with the id of the VMType. For example:

This will deploy your app with the

c2m4

VMType, giving your app 2 CPU cores and 4 GB of RAM.

https://reflex.dev/docs/hosting/regions



To scale your app you can choose different regions. Regions are different locations around the world where your app can be deployed.

To scale your app to multiple regions in the Cloud UI, click on the

Settings

tab in the Cloud UI on the app page, and then click on the

Regions

tab as shown below. Clicking on the

Add new region

button will allow you to scale your app to multiple regions.

The images below show all the regions that can be deployed in.

Selecting Regions to Deploy in the CLI

Below is an example of how to deploy your app in several regions:

By default all apps are deloyed in

sjc

if no other regions are given. If you wish to deploy in another region or several regions you can pass the --region

flag (

-r

also works) with the region code. Check out all the regions that we can deploy to below:

Config File

To create a

config.yml

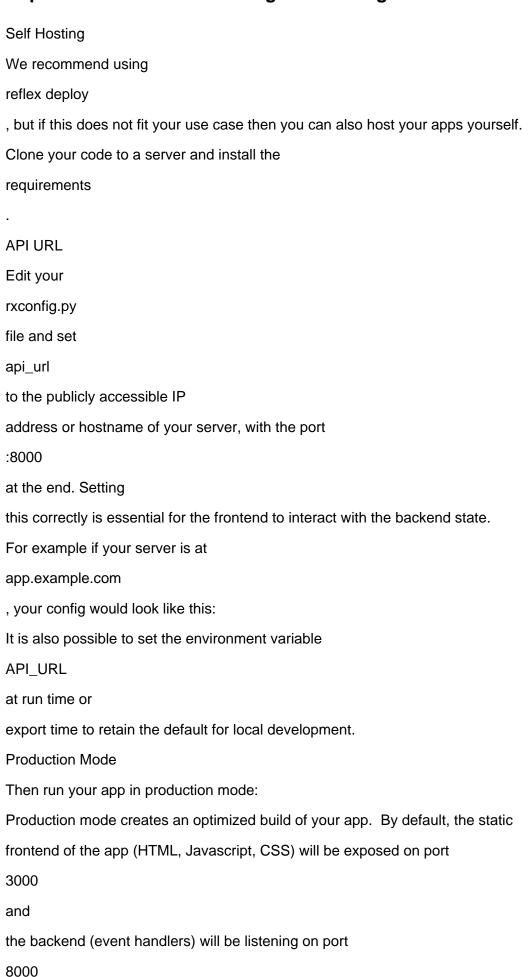
file for your app run the command below:

This will create a yaml file similar to the one below where you can edit the app configuration:

https://reflex.dev/docs/hosting/secrets-environment-vars

Secrets (Environment Variables)
Adding Secrets through the CLI
Below is an example of how to use an environment variable file. You can pass the
envfile
flag with the path to the env file. For example:
In this example the path to the file is
.env
If you prefer to pass the environment variables manually below is deployment command example:
They are passed after the
env
flag as key value pairs.
To pass multiple environment variables, you can repeat the
env
tag. i.e.
reflex deployproject f88b1574-f101-###-###-5f#######env KEY1=VALUE1env KEY2=VALUE
. The
envfile
flag will override any envs set manually.
More information on Environment Variables
Adding Secrets through the Cloud UI
To find the secrets tab, click on the
Settings
tab in the Cloud UI on the app page.
Then click on the
Secrets
tab as shown below.
From here you can add or edit your environment variables. You will need to restart your app for these
changes to take effect.
This functionality in the UI can be disabled by an admin of the project.

https://reflex.dev/docs/hosting/self-hosting



Reverse Proxy and Websockets Exporting a Static Build Exporting a static build of the frontend allows the app to be served using a static hosting provider, like Netlify or Github Pages. Be sure api_url is set to an accessible backend URL when the frontend is exported. This will create a frontend.zip file with your app's minified HTML, Javascript, and CSS build that can be uploaded to your static hosting service. It also creates a backend.zip file with your app's backend python code to upload to your server and run. You can export only the frontend or backend by passing in the --frontend-only or --backend-only flags. It is also possible to export the components without zipping. To do this, use the --no-zip parameter. This provides the frontend in the .web/build/client/ directory and the backend can be found in the root directory of the project. Reflex Container Service Another option is to run your Reflex service in a container. For this purpose, a Dockerfile and additional documentation is available in the Reflex

project in the directory

docker-example
For the build of the container image it is necessary to edit the
rxconfig.py
and the add the
requirements.txt
to your project folder. The following changes are necessary in
rxconfig.py
:
Notice that the
api_url
should be set to the externally accessible hostname or
IP, as the client browser must be able to connect to it directly to establish
interactivity.
You can find the
requirements.txt
in the
docker-example
folder of the
project too.
The project structure should looks like this:
After all changes have been made, the container image can now be created as follows.
Finally, you can start your Reflex container service as follows.

https://reflex.dev/docs/hosting/tokens

Tokens

A token gives someone else all the permissions you have as a User or an Admin. They can run any Reflex Cloud command from the CLI as if they are you using the

--token

flag. A good use case would be for GitHub actions (you store this token in the secrets).

Tokens are found on the Project List page under the tab

Tokens

. If you cannot find it click the Reflex Logo in the top left side of the page until it appears as in the image below.

https://reflex.dev/docs/library

Component Library Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. This page contains a list of all builtin components. Data Display Avatar Badge Callout Code Block Data List Icon List Moment **Progress** Scroll Area Spinner Disclosure Accordion Segmented Control Tabs Dynamic Rendering Auto Scroll Cond Foreach Match Forms **Button** Checkbox Form Input

Slider

Select

Radio Group

Switch
Text Area
Upload
Layout
Aspect Ratio
Вох
Card
Center
Container
Flex
Fragment
Grid
Inset
Section
Separator
Spacer
Stack
Media
Audio
Image
Video
Other
Clipboard
Html
Html Embed
Memo
Script
Skeleton
Theme
Overlay
Alert Dialog
Context Menu
Dialog
Drawer

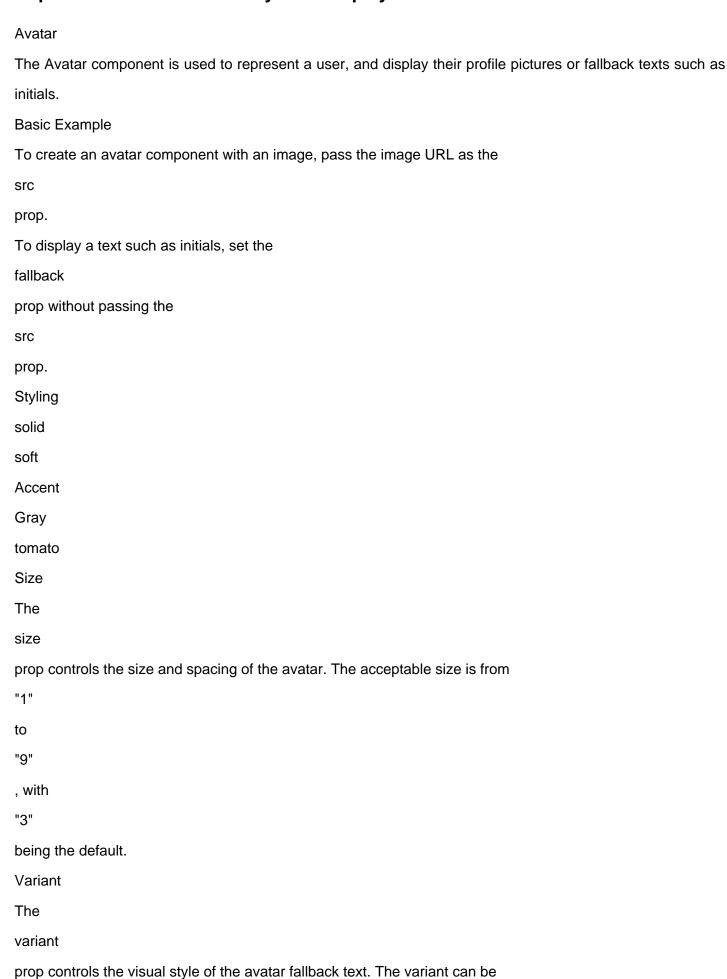
Dropdown Menu
Hover Card
Popover
Toast
Tooltip
Tables And Data Grids
Data Editor
Data Table
Table
Typography
Blockquote
Code
Em
Heading
Kbd
Link
Markdown
Quote
Strong
Text
Graphing Components
Discover our range of components for building interactive charts and data visualizations. Create clear,
informative, and visually engaging representations of your data with ease.
Charts
Areachart
Barchart
Composedchart
Errorbar
Funnelchart
Linechart
Piechart
Radarchart
Radialbarchart
Scatterchart

General
Axis
Brush
Cartesiangrid
Label
Legend
Reference
Tooltip
Other Charts
Plotly
Pyplot

https://reflex.dev/docs/library/data-display

Data Display
Tools to show information clearly. These include ways to highlight important details, show user pictures
display lists, indicate progress, and organize data neatly.
Avatar
Badge
Callout
Code Block
Data List
Icon
List
Moment
Progress
Scroll Area
Spinner

https://reflex.dev/docs/library/data-display/avatar



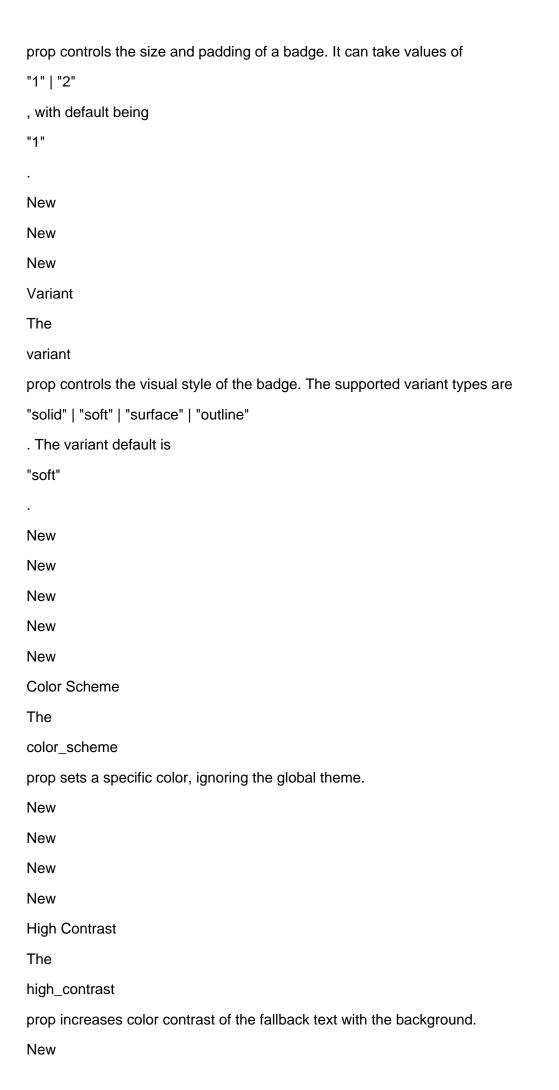
"solid"
or
"soft"
. The default is
"soft"
•
Color Scheme
The
color_scheme
prop sets a specific color to the fallback text, ignoring the global theme.
High Contrast
The
high_contrast
prop increases color contrast of the fallback text with the background.
Radius
The
radius
prop sets specific radius value, ignoring the global theme. It can take values
"none" "small" "medium" "large" "full"
•
Fallback
The
fallback
prop indicates the rendered text when the
src
cannot be loaded.
Final Example
As part of a user profile page, the Avatar component is used to display the user's profile picture, with the
fallback text showing the user's initials. Text components displays the user's full name and username handle
and a Button component shows the edit profile button.
Reflex User
@reflexuser
Edit Profile
API Reference

rx.avatar An image element with a fallback for representing the user. Prop Type | Values Default Interactive variant "solid" | "soft" size "1" | "2" | ... color_scheme "tomato" | "red" | ... tomato high_contrast bool false radius "none" | "small" | ... src str fallback str **Event Triggers**

See the full list of default event triggers

https://reflex.dev/docs/library/data-display/badge

Badge
Badges are used to highlight an item's status for quick recognition.
Basic Example
To create a badge component with only text inside, pass the text as an argument.
New
Styling
solid
soft
surface
outline
Accent
England!
Gray
England!
tomato
Size
The
size



New	
New	
Radius	
The	
radius	
orop sets specific radius value, ignoring the global theme. It can take values	
'none" "small" "medium" "large" "full"	
New	
Final Example	
A badge may contain more complex elements within it. This example uses a	
flex	
component to align an icon and the text correctly, using the	
gap	
orop to	
ensure a comfortable spacing between the two.	
8.8%	
API Reference	
rx.badge	
A stylized badge element.	
Basic Badge	
Prop	
Type Values	
Default	
Interactive	

```
variant

"solid" | "soft" | ...

size

"1" | "2" | ...

color_scheme

"tomato" | "red" | ...

tomato

high_contrast

bool

false

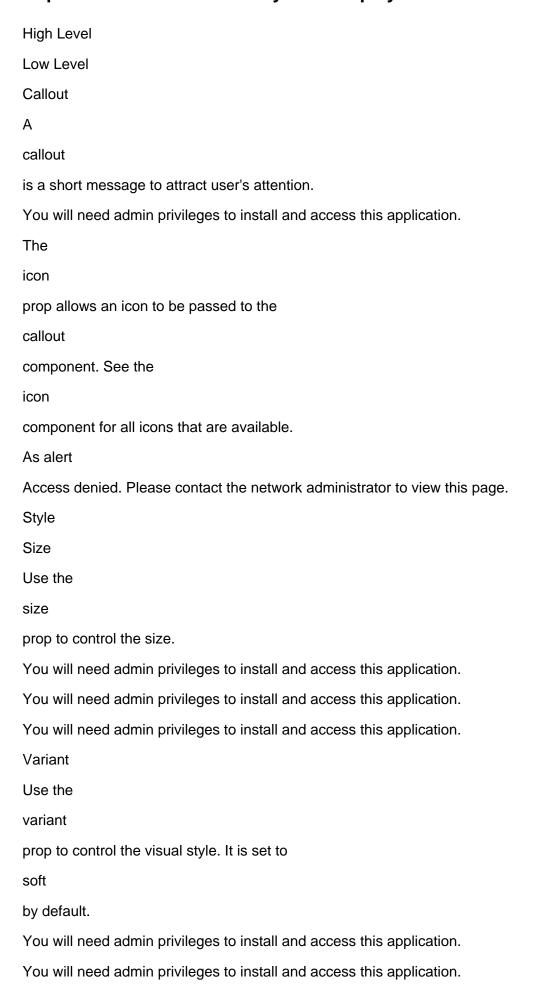
radius

"none" | "small" | ...

Event Triggers
```

See the full list of default event triggers

https://reflex.dev/docs/library/data-display/callout



You will need admin privileges to install and access this application.
Color
Use the
color_scheme
prop to assign a specific color, ignoring the global theme.
You will need admin privileges to install and access this application.
You will need admin privileges to install and access this application.
You will need admin privileges to install and access this application.
High Contrast
Use the
high_contrast
prop to add additional contrast.
You will need admin privileges to install and access this application.
You will need admin privileges to install and access this application.
API Reference
rx.callout
A short message to attract user's attention.
Basic Callout
Prop
Type Values
Default
Interactive
text
str
icon
str
as_child
bool
size
"1" "2"
variant
"soft" "surface"
color_scheme
"tomato" "red"

tomato
high_contrast
bool
false
Event Triggers
See the full list of default event triggers
rx.callout.root
Groups Icon and Text parts of a Callout.
You will need admin privileges to install and access this application.
Prop
Type Values
Default
Interactive
as_child
bool
size
"1" "2"
variant
"soft" "surface"
color_scheme
"tomato" "red"
tomato
high_contrast
bool
false
Event Triggers
See the full list of default event triggers
rx.callout.icon
Provides width and height for the icon associated with the callout.
Props
No component specific props
Event Triggers
See the full list of default event triggers
rx.callout.text

Renders the callout text. This component is based on the p element.

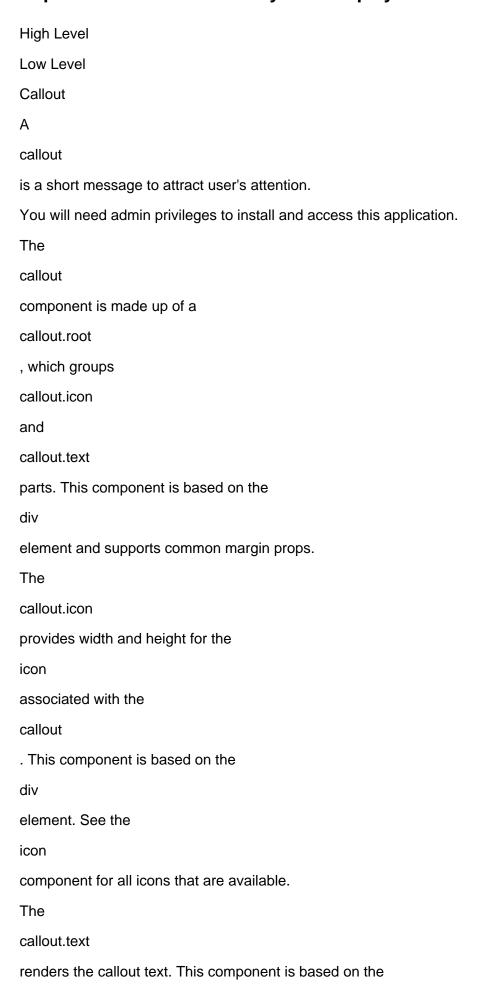
Props

No component specific props

Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/data-display/callout/low



p
element.
As alert
Access denied. Please contact the network administrator to view this page.
Style
Size
Use the
size
prop to control the size.
You will need admin privileges to install and access this application.
You will need admin privileges to install and access this application.
You will need admin privileges to install and access this application.
Variant
Use the
variant
prop to control the visual style. It is set to
soft
by default.
You will need admin privileges to install and access this application.
You will need admin privileges to install and access this application.
You will need admin privileges to install and access this application.
Color
Use the
color_scheme
prop to assign a specific color, ignoring the global theme.
You will need admin privileges to install and access this application.
You will need admin privileges to install and access this application.
You will need admin privileges to install and access this application.
High Contrast
Use the
high_contrast
prop to add additional contrast.
You will need admin privileges to install and access this application.
You will need admin privileges to install and access this application.

API Reference
rx.callout
A short message to attract user's attention.
Basic Callout
Prop
Type Values
Default
Interactive
text
str
icon
str
as_child
bool
size
"1" "2"
variant
"soft" "surface"
color_scheme
"tomato" "red"
tomato
high_contrast
bool
false
Event Triggers
See the full list of default event triggers
rx.callout.root
Groups Icon and Text parts of a Callout.
You will need admin privileges to install and access this application.
Prop
Type Values
Default
Interactive
as child

https://reflex.dev/docs/library/data-display/code-block

Code Block

The Code Block component can be used to display code easily within a website.

Put in a multiline string with the correct spacing and specify and language to show the desired code.

```
1
2
3
4
5
def fib(n):
  if n <= 1:
     return n
  else:
     return(fib(n-1) + fib(n-2))
API Reference
rx.code_block
A code block.
Prop
Type | Values
Default
theme
Union[Theme, str]
Theme.one_light
language
"abap" | "abnf" | ...
Var.create("python")
code
str
show_line_numbers
bool
starting_line_number
int
wrap_long_lines
```

bool

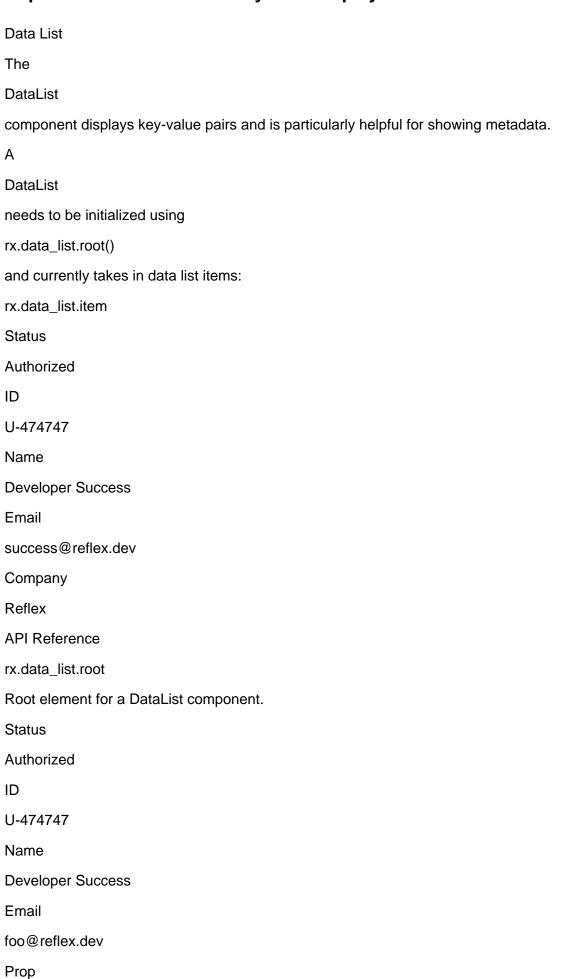
code_tag_props

Dict[str, str]

Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/data-display/data-list



Type Values
Default
Interactive
orientation
"horizontal" "vertical"
size
"1" "2"
trim
"normal" "start"
Event Triggers
See the full list of default event triggers
rx.data_list.item
An item in the DataList component.
Status
Authorized
ID
U-474747
Name
Developer Success
Email
foo@reflex.dev
Prop
Type Values
Default
Interactive
align
"start" "center"
Event Triggers
See the full list of default event triggers
rx.data_list.label
A label in the DataList component.
Status
Authorized

ID

Na	ame
De	eveloper Success
En	nail
foc	@reflex.dev
Pr	ор
Ту	pe Values
De	efault
Int	eractive
wie	dth
str	
mi	n_width
str	
ma	ax_width
str	
со	lor_scheme
"to	omato" "red"
tor	mato
Εv	vent Triggers
Se	ee the full list of default event triggers
rx.	data_list.value
Α١	value in the DataList component.
Pr	ops
No	component specific props
Εv	vent Triggers
Se	ee the full list of default event triggers

https://reflex.dev/docs/library/data-display/icon

prop in

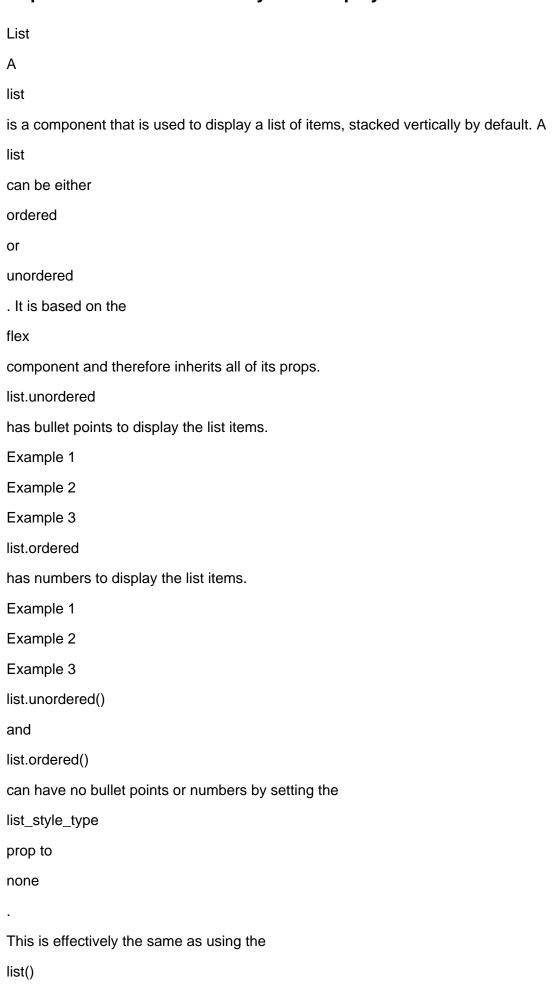
Icon The Icon component is used to display an icon from a library of icons. This implementation is based on the Lucide Icons where you can find a list of all available icons. Icons List Show all (0) **Basic Example** To display an icon, specify the tag prop from the list of available icons. Passing the tag as the first children is also supported and will be assigned to the tag prop. The tag is expected to be in snake_case format, but kebab-case is also supported to allow copy-paste from https://lucide.dev/icons Dynamic Icons There are two ways to use dynamic icons in Reflex: Using rx.match If you have a specific subset of icons you want to use dynamically, you can define an rx.match with them: Using Dynamic Icon Tags Reflex also supports using dynamic values directly as the tag

rx.icon() . This allows you to use any icon from the Lucide library dynamically at runtime. Dynamic Icon Example Change Icon Under the hood, when a dynamic value is passed as the tag prop to rx.icon() , Reflex automatically uses a special DynamicIcon component that can load icons at runtime. When using dynamic icons, make sure the icon names are valid. Invalid icon names will cause runtime errors. Styling Icon from Lucide can be customized with the following props stroke_width size and color Stroke Width Size Color Here is an example using basic colors in icons. A radix color with a scale may also be specified using rx.color() as seen below. Here is another example using the accent color with scales. The accent is the most dominant color in your theme.

Final Example

Icons can be used as child components of many other components. For example, adding a magnifying glass
icon to a search bar.
Search documentation
API Reference
rx.lucide.lcon
An Icon component.
Prop
Type Values
Default
size
int
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/data-display/list



component.
Example 1
Example 2
Example 3
Example 1
Example 2
Example 3
Lists can also be used with icons.
Allowed
Not
Settings
API Reference
rx.list.item
Display an item of an ordered or unordered list.
Props
No component specific props
Event Triggers
See the full list of default event triggers
rx.list.ordered
Display an ordered list.
Prop
Type Values
Default
list_style_type
"none" "disc"
items
Iterable
Var.create([])
Event Triggers
See the full list of default event triggers
rx.list.unordered
Display an unordered list.
Prop

Type | Values

Default

list_style_type

"none" | "disc" | ...

items

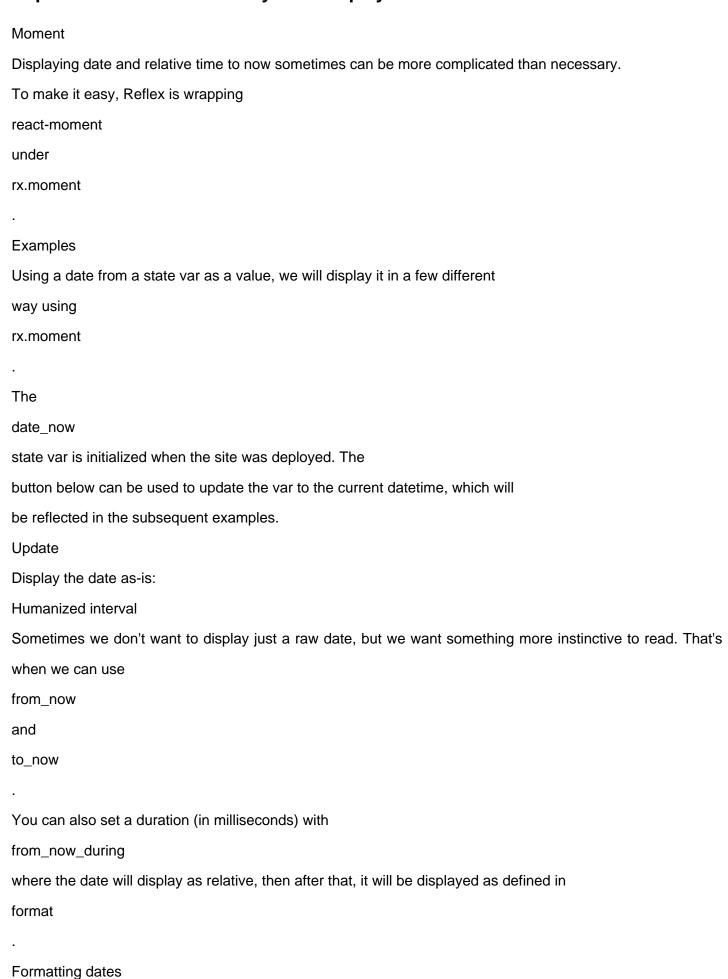
Iterable

Var.create([])

Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/data-display/moment



Offset Date
With the props
add
and
subtract
, you can pass an
rx.MomentDelta
object to modify the displayed date without affecting the stored date in your state.
Add
Add
Subtract
Subtract
Timezones
You can also set dates to display in a specific timezone:
Client-side periodic update
If a date is not passed to
rx.moment
, it will use the client's current time.
If you want to update the date every second, you can use the
interval
prop.
Even better, you can actually link an event handler to the
on_change
prop that will be called every time the date is updated:
API Reference
rx.moment
The Moment component.
Prop
Type Values
Default
interval
int
format
str

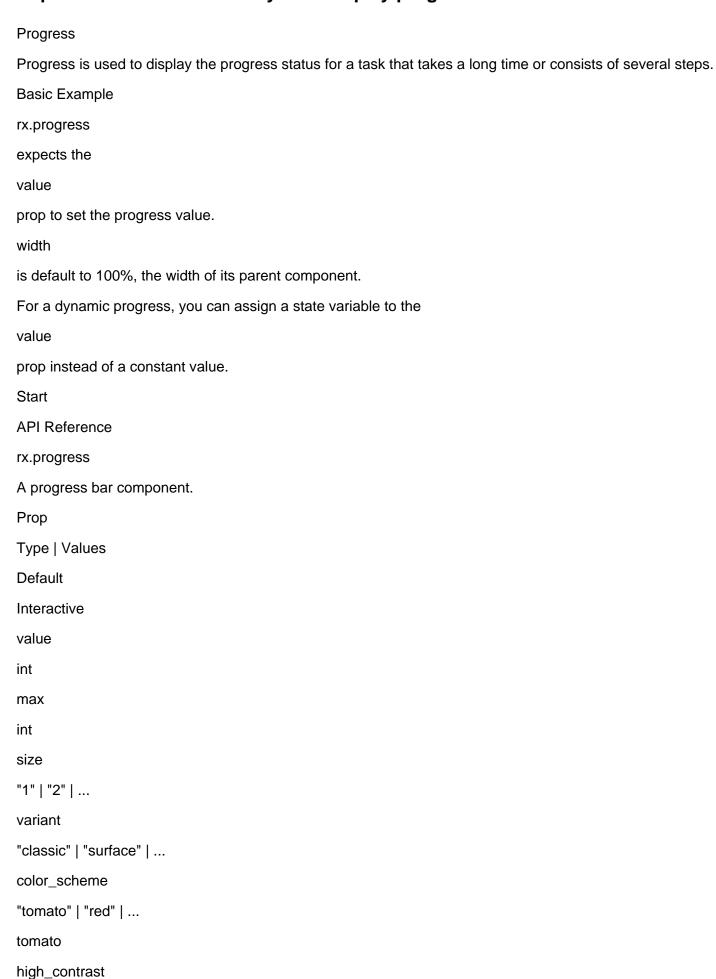
trim
bool
parse
str
add
MomentDelta
subtract
MomentDelta
from_now
bool
from_now_during
int
to_now
bool
with_title
bool
title_format
str
diff
str
decimal
bool
unit
str
duration
str
date
Union[str, datetime, date, time, timedelta]
duration_from_now
bool
unix
bool
local
bool

tz
str
locale
str
Event Triggers
See the full list of default event triggers
Trigger
Description

Fires when the date changes.

on_change

https://reflex.dev/docs/library/data-display/progress



bool
false
radius
"none" "small"
duration
str
fill_color
str
Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/data-display/scroll-area

Scroll Area

Custom styled, cross-browser scrollable area using native functionality.

Basic Example

Three fundamental aspects of typography are legibility, readability, and aesthetics. Although in a non-technical sense "legible― and "readable― are often used synonymously, typographically they are separate but related concepts.

Legibility describes how easily individual characters can be distinguished from one another. It is described by Walter Tracy as "the quality of being decipherable and recognisable―. For instance, if a "b― and an "h―, or a "3― and an "8―, are difficult to distinguish at small sizes, this is a problem of legibility.

Typographers are concerned with legibility insofar as it is their job to select the correct font to use. Brush Script is an example of a font containing many characters that might be difficult to distinguish. The selection of cases influences the legibility of typography because using only uppercase letters (all-caps) reduces legibility.

Control the scrollable axes

Use the

scrollbars

prop to limit scrollable axes. This prop can take values

"vertical" | "horizontal" | "both"

Three fundamental aspects of typography are legibility, readability, and aesthetics. Although in a non-technical sense "legible" and "readable" are often used synonymously, typographically they are separate but related concepts.

Legibility describes how easily individual characters can be distinguished from one another. It is described by Walter Tracy as "the quality of being decipherable and recognisable". For instance, if a "b" and an "h", or a "3" and an "8", are difficult to distinguish at small sizes, this is a problem of legibility.

Three fundamental aspects of typography are legibility, readability, and aesthetics. Although in a non-technical sense "legible" and "readable" are often used synonymously, typographically they are separate but related concepts.

Legibility describes how easily individual characters can be distinguished from one another. It is described by Walter Tracy as "the quality of being decipherable and recognisable". For instance, if a "b" and an "h", or a "3" and an "8", are difficult to distinguish at small sizes, this is a problem of legibility.

Three fundamental aspects of typography are legibility, readability, and aesthetics. Although in a non-technical sense "legible" and "readable" are often used synonymously, typographically they are separate but related concepts.

Legibility describes how easily individual characters can be distinguished from one another. It is described by Walter Tracy as "the quality of being decipherable and recognisable". For instance, if a "b" and an "h", or a "3" and an "8", are difficult to distinguish at small sizes, this is a problem of legibility.

Setting the type of the Scrollbars

The

type

prop describes the nature of scrollbar visibility.

auto

means that scrollbars are visible when content is overflowing on the corresponding orientation.

always

means that scrollbars are always visible regardless of whether the content is overflowing.

scroll

means that scrollbars are visible when the user is scrolling along its corresponding orientation.

hover

when the user is scrolling along its corresponding orientation and when the user is hovering over the scroll area.

type = 'auto'

Legibility describes how easily individual characters can be distinguished from one another. It is described by Walter Tracy as "the

quality of being decipherable and recognisable". For instance, if a "b" and an "h", or a "3" and an "8", are difficult to distinguish at small sizes, this is a problem of legibility.

type = 'always'

Legibility describes how easily individual characters can be distinguished from one another. It is described by Walter Tracy as "the quality of being decipherable and recognisable". For instance, if a "b" and an "h", or a "3" and an "8", are difficult to distinguish at small sizes, this is a problem of legibility.

type = 'scroll'

Legibility describes how easily individual characters can be distinguished from one another. It is described by Walter Tracy as "the quality of being decipherable and recognisable". For instance, if a "b" and an "h", or a "3" and an "8", are difficult to distinguish at small sizes, this is a problem of legibility.

type = 'hover'

Legibility describes how easily individual characters can be distinguished from one another. It is described by Walter Tracy as "the quality of being decipherable and recognisable". For instance, if a "b" and an "h", or a "3" and an "8", are difficult to distinguish at small sizes, this is a problem of legibility.

API Reference

rx.scroll_area

Custom styled, cross-browser scrollable area using native functionality.

Three fundamental aspects of typography are legibility, readability, and aesthetics. Although in a non-technical sense "legible" and "readable" are often used synonymously, typographically they are separate but related concepts.

Legibility describes how easily individual characters can be distinguished from one another. It is described by Walter Tracy as "the quality of being decipherable and recognisable". For instance, if a "b" and an "h", or a "3" and an "8", are difficult to distinguish at small sizes, this is a problem of legibility.

Prop

Type | Values

Default

Interactive

scrollbars

"vertical" | "horizontal" | ...

type

"auto" | "always" | ...

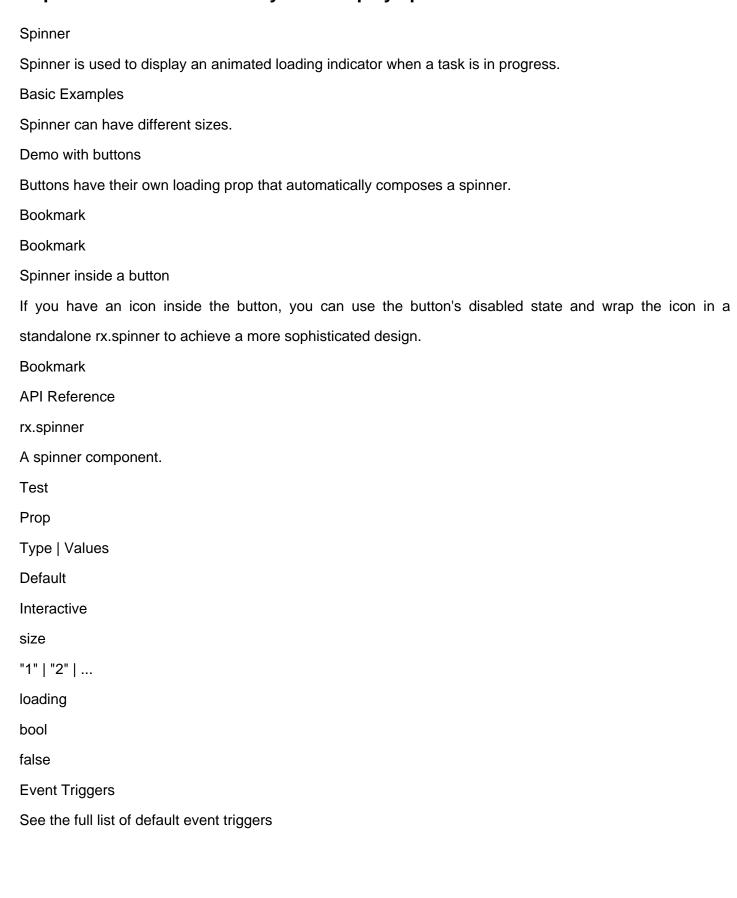
scroll_hide_delay

int

Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/data-display/spinner



https://reflex.dev/docs/library/disclosure

Disclosure

Components for revealing or hiding content, such as tabs and accordions. These are useful for creating expandable sections, organizing information, and improving user interface navigation.

Accordion

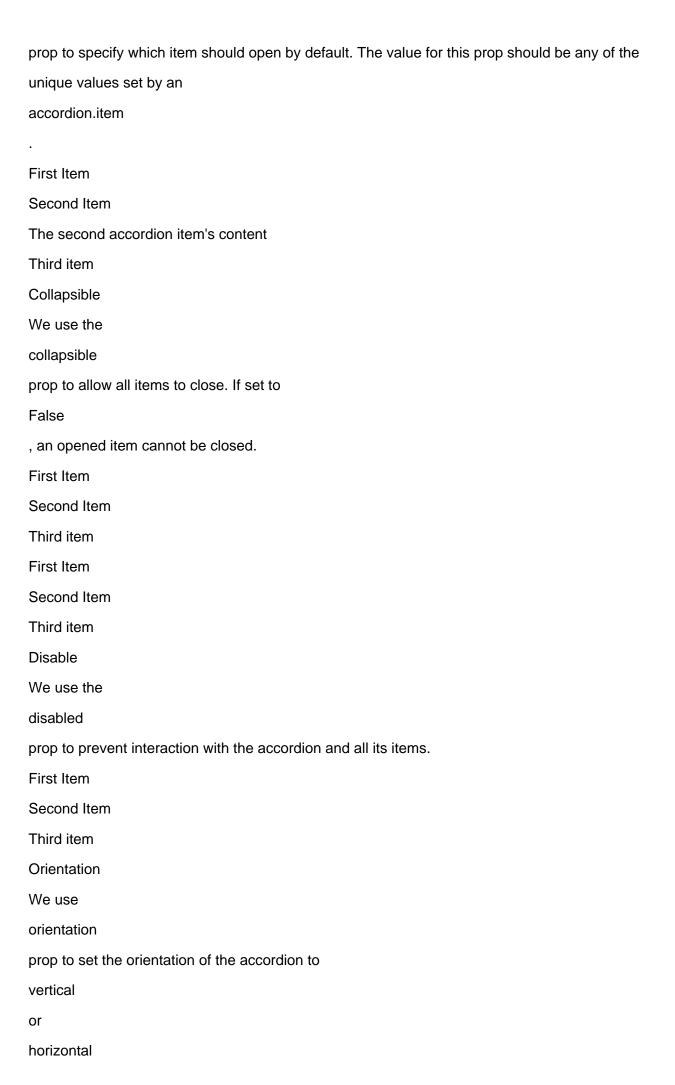
Segmented Control

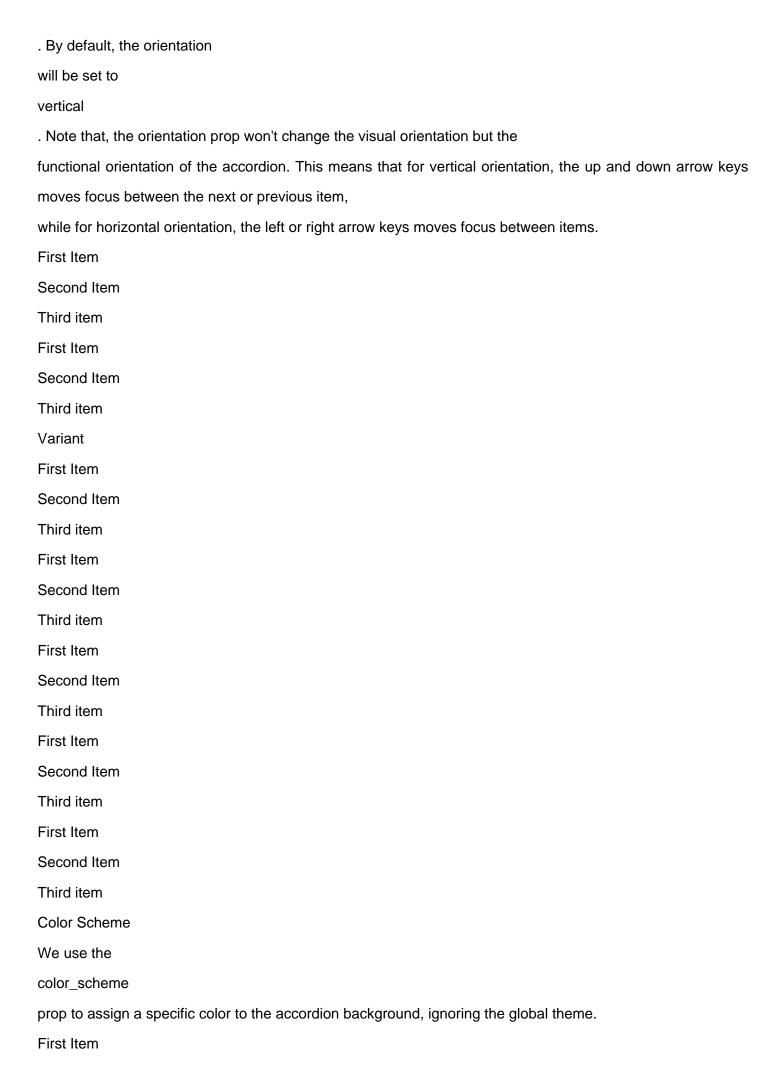
Tabs

https://reflex.dev/docs/library/disclosure/accordion

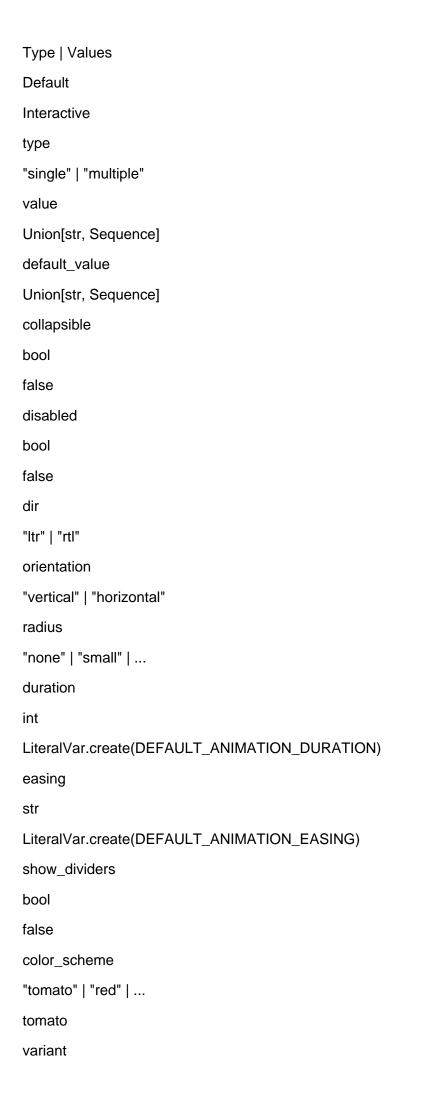
default_value

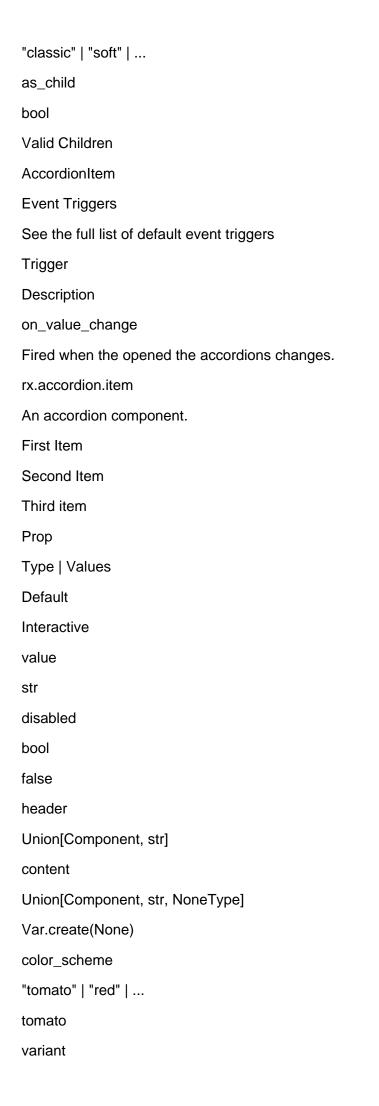
Accordion An accordion is a vertically stacked set of interactive headings that each reveal an associated section of content. The accordion component is made up of accordion , which is the root of the component and takes in an accordion.item which contains all the contents of the collapsible section. Basic Example First Item Second Item Third item Styling Type We use the type prop to determine whether multiple items can be opened at once. The allowed values for this prop are single and multiple where single will only open one item at a time. The default value for this prop is single First Item Second Item Third item **Default Value** We use the





Second Item
Third item
First Item
Second Item
Third item
Value
We use the
value
prop to specify the controlled value of the accordion item that we want to activate.
This property should be used in conjunction with the
on_value_change
event argument.
Is it accessible?
Test button
Is it unstyled?
Is it finished?
AccordionItem
The accordion item contains all the parts of a collapsible section.
Styling
Value
First Item
Second Item
Third item
Disable
First Item
Second Item
Third item
API Reference
rx.accordion.root
An accordion component.
First Item
Second Item
Third item
Prop





"classic" | "soft" | ...
as_child
bool
Valid Children
AccordionHeader
AccordionTrigger

AccordionContent

Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/disclosure/segmented-control

Segmented Control Segmented Control offers a clear and accessible way to switch between predefined values and views, e.g., "Inbox," "Drafts," and "Sent." With Segmented Control, you can make mutually exclusive choices, where only one option can be active at a time, clear and accessible. Without Segmented Control, end users might have to deal with controls like dropdowns or multiple buttons that don't clearly convey state or group options together visually. Basic Example The Segmented Control component is made up of a rx.segmented_control.root which groups rx.segmented_control.item The rx.segmented_control.item components define the individual segments of the control, each with a label and a unique value. Home Home About About Test Test test test test In the example above: on_change is used to specify a callback function that will be called when the user selects a different segment. In this case, the

SegmentedState.setvar("control")

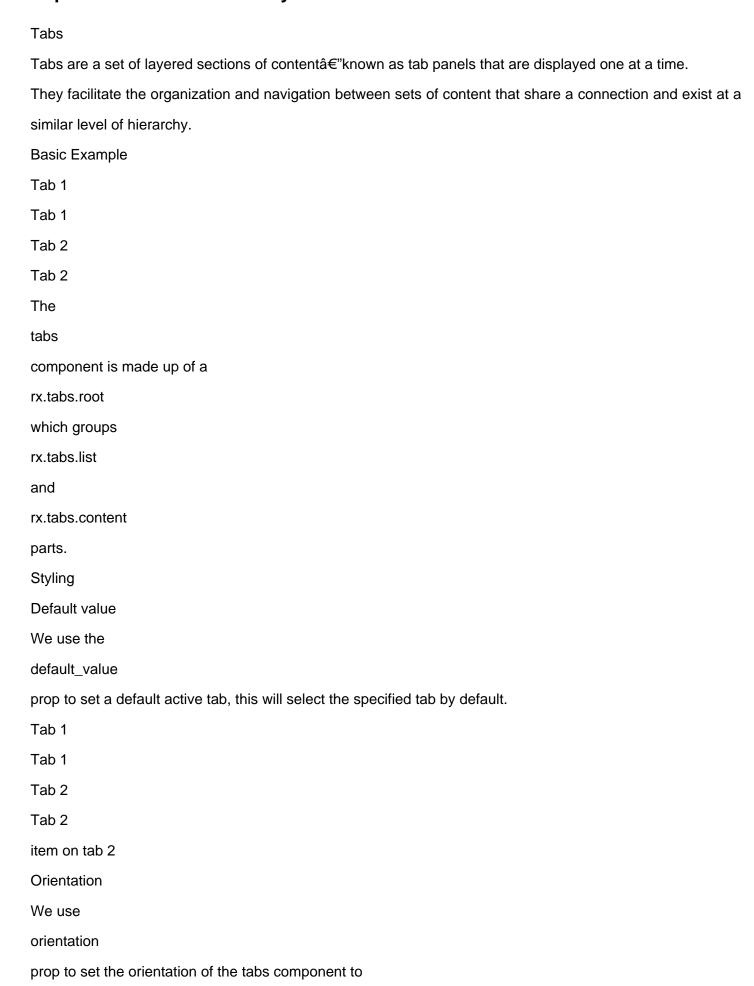
function is used to update the

control state variable when the user changes the selected segment. value prop is used to specify the currently selected segment, which is bound to the SegmentedState.control state variable. **API Reference** rx.segmented_control.root Root element for a SegmentedControl component. Test Prop Type | Values Default Interactive size "1" | "2" | ... variant "classic" | "surface" type "single" | "multiple" color_scheme "tomato" | "red" | ... tomato radius "none" | "small" | ... default_value Union[str, Sequence] value Union[str, Sequence] **Event Triggers** See the full list of default event triggers Trigger Description

on_change

Handles the `onChange` event for the SegmentedControl component.
rx.segmented_control.item
An item in the SegmentedControl component.
Prop
Type Values
Default
Interactive
value
str
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/disclosure/tabs

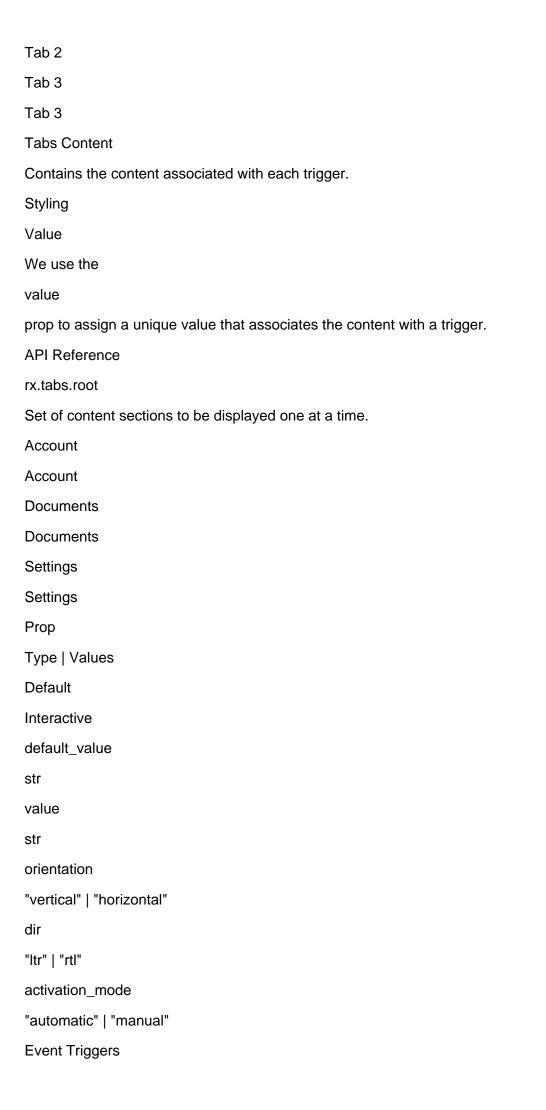


vertical
or
horizontal
. By default, the orientation
will be set to
horizontal
. Setting this value will change both the visual orientation and the functional orientation.
The functional orientation means for
vertical
, the
up
and
down
arrow keys moves focus between the next or previous tab,
while for
horizontal
, the
left
and
right
arrow keys moves focus between tabs.
When using vertical orientation, make sure to assign a tabs.content for each trigger.
Tab 1
Tab 1
Tab 2
Tab 2
item on tab 1
Tab 1
Tab 1
Tab 2
Tab 2
item on tab 1
Value
We use the

value
prop to specify the controlled value of the tab that we want to activate. This property should be used
conjunction with the
on_change
event argument.
tab1 clicked!
Tab 1
Tab 1
Tab 2
Tab 2
items on tab 1
Tablist
The Tablist is used to list the respective tabs to the tab component
Tab Trigger
This is the button that activates the tab's associated content. It is typically used in the
Tablist
Styling
Value
We use the
value
prop to assign a unique value that associates the trigger with content.
Tab 1
Tab 1
Tab 2
Tab 2
Tab 3
Tab 3
Disable
We use the
disabled
prop to disable the tab.
Tab 1
Tab 1

Tab 2

in



See the full list of default event triggers
Trigger
Description
on_change
Props to rename Fired when the value of the tabs changes.
rx.tabs.list
Contains the triggers that sit alongside the active content.
Account
Account
Documents
Documents
Settings
Settings
Prop
Type Values
Default
Interactive
size
"1" "2"
loop
bool
false
Event Triggers
See the full list of default event triggers
rx.tabs.trigger
The button that activates its associated content.
Account
Account
Documents
Documents
Settings
Settings
Prop
Type Values

Default
Interactive
value
str
disabled
bool
false
color_scheme
"tomato" "red"
tomato
Event Triggers
See the full list of default event triggers
rx.tabs.content
Contains the content associated with each trigger.
Account
Account
Documents
Documents
Settings
Settings
Prop
Type Values
Default
Interactive
value
str
force_mount
bool
false
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/forms

Forms
Components for collecting user input, such as text fields, checkboxes, and radio buttons. These are useful for
creating interactive forms and user input.
Button
Checkbox
Form
Input
Radio Group
Select
Slider
Switch
Text Area
Upload

https://reflex.dev/docs/library/forms/input



required
prop is
True
, it indicates that the user must input text before the owning form can be submitted.
The
type
is set here to
password
. The element is presented as a one-line plain text editor control in which the text is obscured so that it cannot
be read. The
type
prop can take any value of
email
,
file
,
password
,
text
and several others. Learn more
here
Example Form
Submit
Results:
{}
To learn more about how to use forms in the
Form
docs.
Setting a value without using a State var
Set the value of the specified reference element, without needing to link it up to a State var. This is an
alternate way to modify the value of the
input

Erase
API Reference
rx.input
Captures user input with an optional slot for buttons and icons.
Test
Prop
Type Values
Default
Interactive
size
"1" "2"
variant
"classic" "surface"
color_scheme
"tomato" "red"
tomato
radius
"none" "small"
auto_complete
bool
false
default_value
str
disabled
bool
false
max_length
int
min_length
int
name
str
placeholder

str

read_only
bool
false
required
bool
false
type
str
value
Union[str, int, float]
list
str
accept
str
alt
str
auto_focus
bool
false
capture
"True" "False"
checked
bool
default_checked
bool
form
str
form_action
str
form_enc_type
str
form_method
str
form_no_validate

bool
false
form_target
str
max
Union[str, int, float]
min
Union[str, int, float]
multiple
bool
false
pattern
str
src
str
step
Union[str, int, float]
Event Triggers
See the full list of default event triggers
Trigger
Description
on_focus
Fired when the textarea is focused.
on_blur
Fired when the textarea is blurred.
on_key_down
Fired when a key is pressed down.
on_key_up
Fired when a key is released.
on_change
Fired when the value of the textarea changes.
rx.input.slot
Contains icons or buttons associated with an Input.
Prop

Type | Values

Default
Interactive

color_scheme
"tomato" | "red" | ...

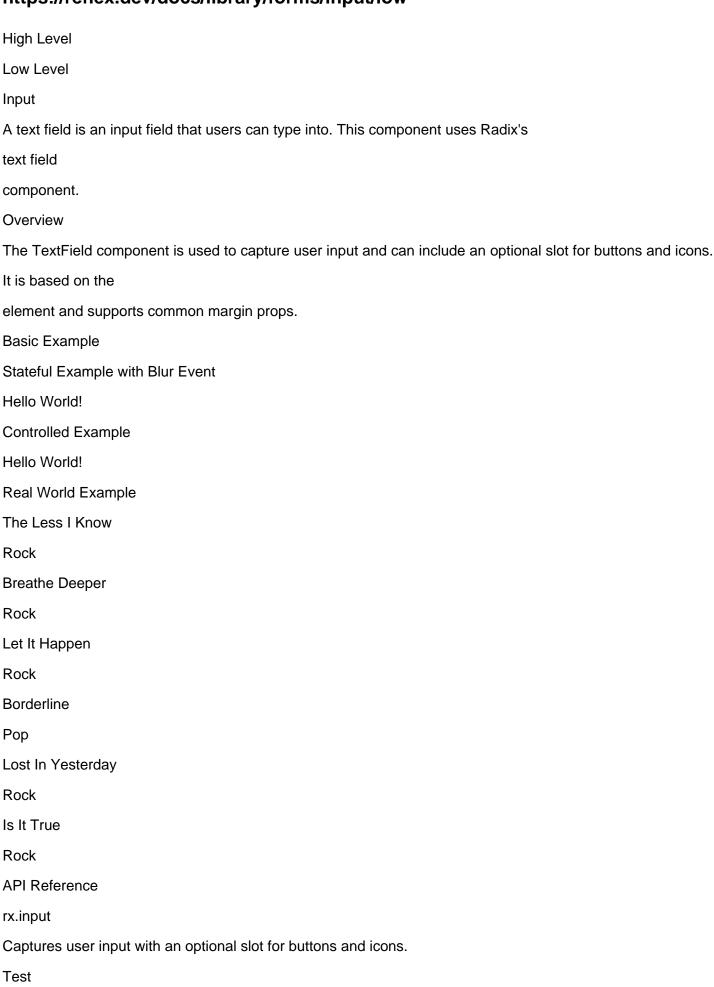
tomato

side
"left" | "right"

Event Triggers

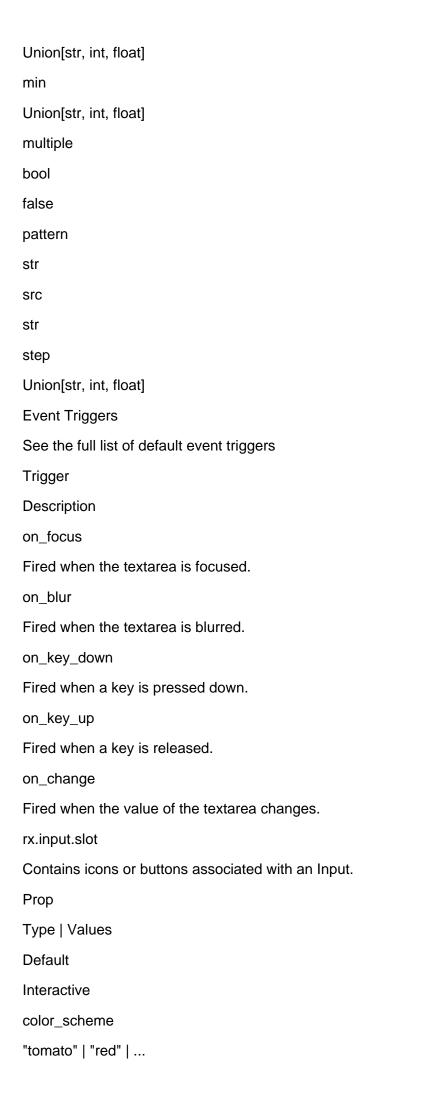
See the full list of default event triggers

https://reflex.dev/docs/library/forms/input/low



Prop
Type Values
Default
Interactive
size
"1" "2"
variant
"classic" "surface"
color_scheme
"tomato" "red"
tomato
radius
"none" "small"
auto_complete
bool
false
default_value
str
disabled
bool
false
max_length
int
min_length
int
name
str
placeholder
str
read_only
bool
false
required
bool

false
type
str
value
Union[str, int, float]
list
str
accept
str
alt
str
auto_focus
bool
false
capture
"True" "False"
checked
bool
default_checked
bool
form
str
form_action
str
form_enc_type
str
form_method
str
form_no_validate
bool
false
form_target
str
max



tomato

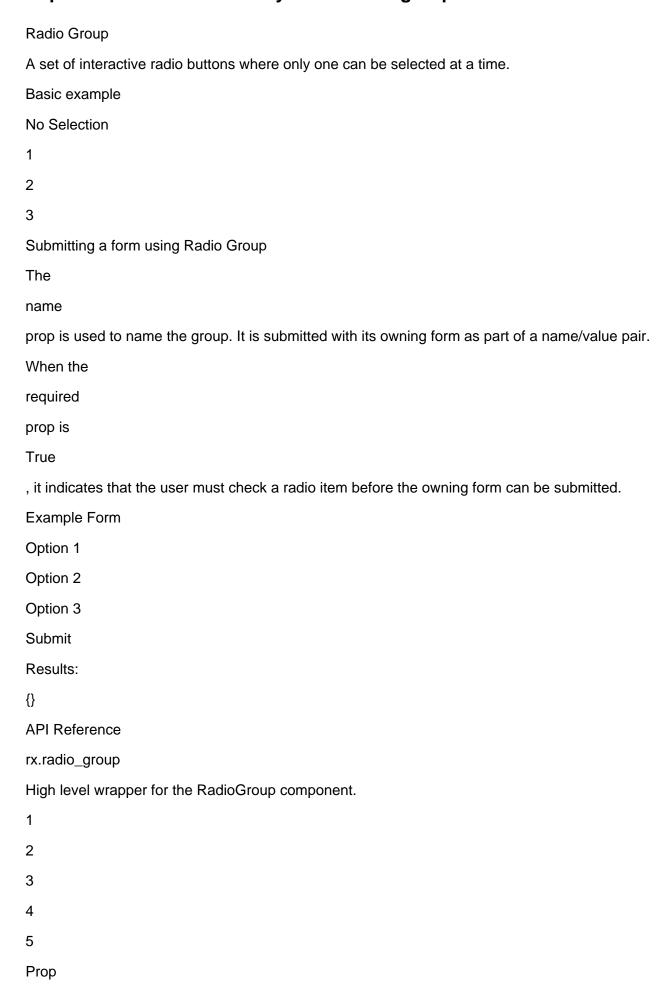
side

"left" | "right"

Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/forms/radio-group

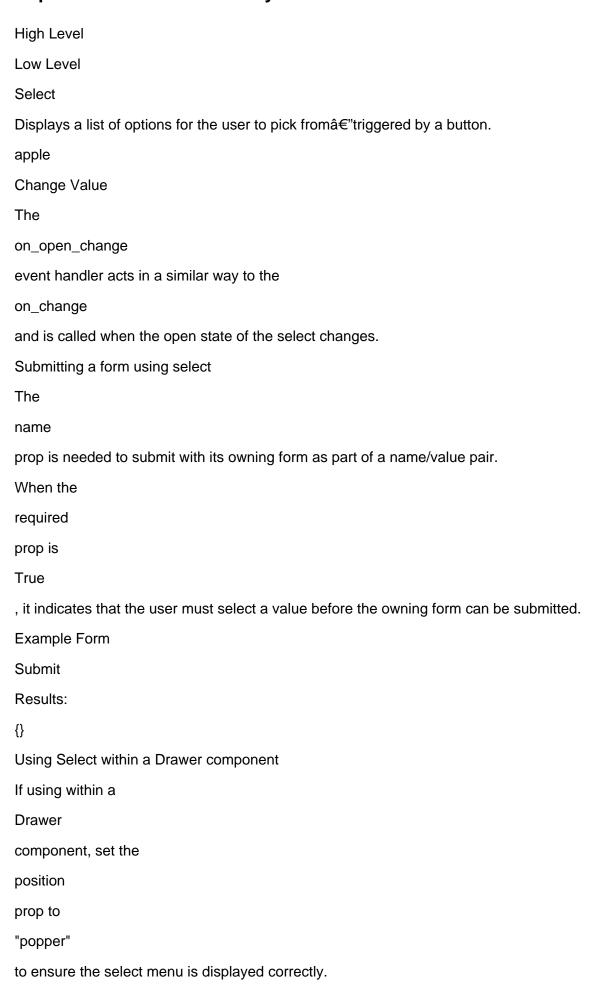


Type Values
Default
Interactive
items
Sequence
direction
"row" "column"
LiteralVar.create("row")
spacing
"0" "1"
LiteralVar.create("2")
size
"1" "2"
LiteralVar.create("2")
variant
"classic" "surface"
LiteralVar.create("classic")
color_scheme
"tomato" "red"
tomato
high_contrast
bool
false
value
str
default_value
str
disabled
bool
false
name
str
required
bool

false
Event Triggers
See the full list of default event triggers
rx.radio_group.root
A set of interactive radio buttons where only one can be selected at a time.
Prop
Type Values
Default
Interactive
size
"1" "2"
LiteralVar.create("2")
variant
"classic" "surface"
LiteralVar.create("classic")
color_scheme
"tomato" "red"
tomato
high_contrast
bool
false
value
str
default_value
str
disabled
bool
false
name
str
required
bool
false
Event Triggers

See the full list of default event triggers
Trigger
Description
on_change
Props to rename Fired when the value of the radio group changes.
rx.radio_group.item
An item in the group that can be checked.
Prop
Type Values
Default
Interactive
value
str
disabled
bool
false
required
bool
false
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/forms/select



Open Drawer
API Reference
rx.select
High level wrapper for the Select component.
Prop
Type Values
Default
Interactive
items
Sequence
placeholder
str
label
str
color_scheme
"tomato" "red"
tomato
high_contrast
bool
false
variant
"classic" "surface"
radius
"none" "small"
width
str
position
"item-aligned" "popper"
size
"1" "2"
default_value
str
value

str

default_open
bool
open
bool
name
str
disabled
bool
false
required
bool
false
Event Triggers
See the full list of default event triggers
Trigger
Description
on_change
Props to rename Fired when the value of the select changes.
on_open_change
Fired when the select is opened or closed.
rx.select.root
Displays a list of options for the user to pick from, triggered by a button.
Prop
Type Values
Default
Interactive
size
"1" "2"
default_value
str
value
str
default_open

open
bool
name
str
disabled
bool
false
required
bool
false
Event Triggers
See the full list of default event triggers
Trigger
Description
on_change
Props to rename Fired when the value of the select changes.
on_open_change
Fired when the select is opened or closed.
rx.select.trigger
The button that toggles the select.
Prop
Type Values
Default
Interactive
variant
"classic" "surface"
color_scheme
"tomato" "red"
tomato
radius
"none" "small"
placeholder
str
Event Tringers

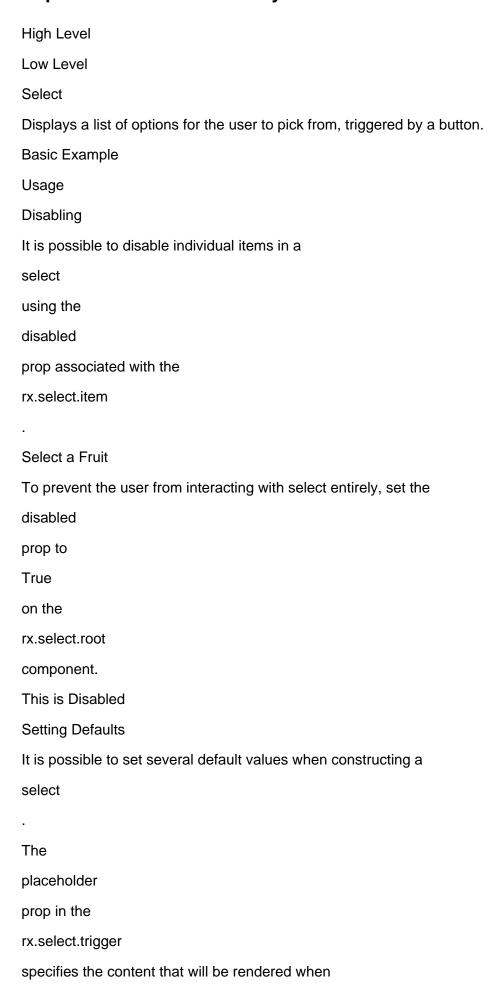
Event Triggers

See the full list of default event triggers rx.select.content The component that pops out when the select is open. Prop Type | Values Default Interactive variant "solid" | "soft" color_scheme "tomato" | "red" | ... tomato high_contrast bool false position "item-aligned" | "popper" side "top" | "right" | ... side_offset int align "start" | "center" | ... align_offset int **Event Triggers** See the full list of default event triggers Trigger Description on_close_auto_focus Fired when the select content is closed. on_escape_key_down Fired when the escape key is pressed.

on_pointer_down_outside

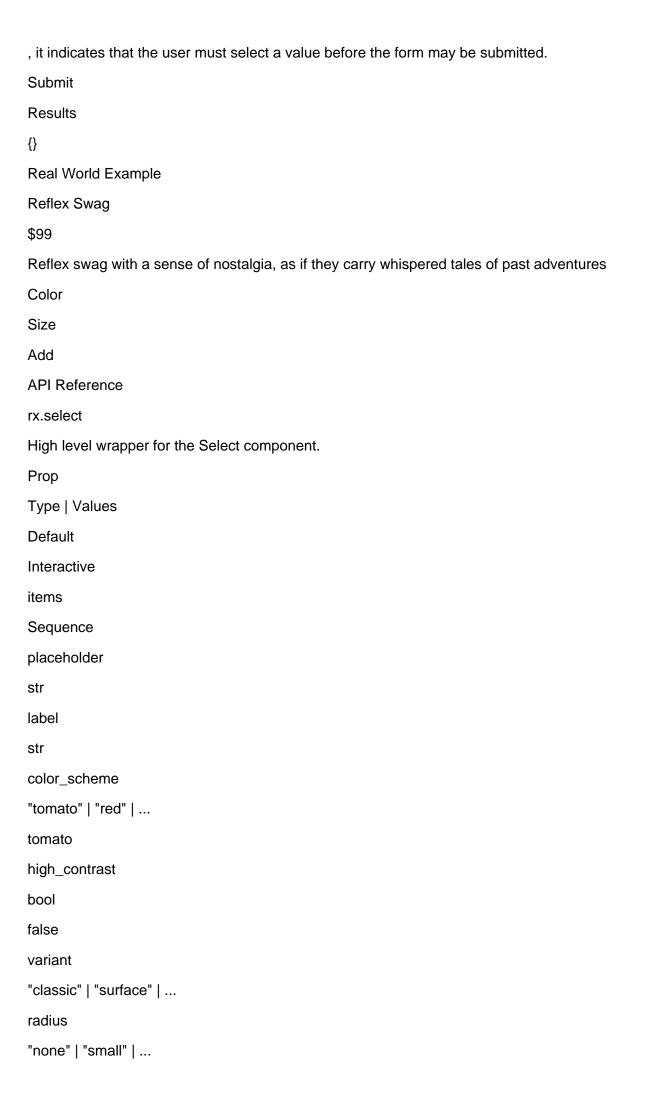
Fired when a pointer down event happens outside the select content.
rx.select.group
Used to group multiple items.
Props
No component specific props
Event Triggers
See the full list of default event triggers
rx.select.item
The component that contains the select items.
Prop
Type Values
Default
Interactive
value
str
disabled
bool
false
Event Triggers
See the full list of default event triggers
rx.select.label
Used to render the label of a group, it isn't focusable using arrow keys.
Props
No component specific props
Event Triggers
See the full list of default event triggers
rx.select.separator
Used to visually separate items in the Select.
Props
No component specific props
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/forms/select/low



value
or
default_value
is empty or not set.
pick a fruit
The
default_value
in the
rx.select.root
specifies the value of the
select
when initially rendered.
The
default_value
should correspond to the
value
of a child
rx.select.item
Fully controlled
The
on_change
event trigger is fired when the value of the select changes.
In this example the
rx.select_root
value
prop specifies which item is selected, and this
can also be controlled using state and a button without direct interaction with the select component.
No Selection
Choose Randomly
Reset
The
open
open

```
on_open_change
event trigger work similarly to
value
and
on_change
to control the open state of the select.
lf
on_open_change
handler does not alter the
open
prop, the select will not be able to be opened or closed by clicking on the
select_trigger
No Selection
Toggle
Submitting a Form with Select
When a select is part of a form, the
name
prop of the
rx.select.root
sets the key that will be submitted with the form data.
The
value
prop of
rx.select.item
provides the value to be associated with the
name
key when the form is submitted with that item selected.
When the
required
prop of the
rx.select.root
is
True
```



width
str
position
"item-aligned" "popper"
size
"1" "2"
default_value
str
value
str
default_open
bool
open
bool
name
str
disabled
bool
false
required
bool
false
Event Triggers
See the full list of default event triggers
Trigger
Description
on_change
Props to rename Fired when the value of the select changes.
on_open_change
Fired when the select is opened or closed.
rx.select.root
Displays a list of options for the user to pick from, triggered by a button.
Prop
Type Values

Default
Interactive
size
"1" "2"
default_value
str
value
str
default_open
bool
open
bool
name
str
disabled
bool
false
required
bool
false
Event Triggers
See the full list of default event triggers
Trigger
Description
on_change
Props to rename Fired when the value of the select changes.
on_open_change
Fired when the select is opened or closed.
rx.select.trigger
The button that toggles the select.
Prop
Type Values
Default
Interactive

```
variant
"classic" | "surface" | ...
color_scheme
"tomato" | "red" | ...
tomato
radius
"none" | "small" | ...
placeholder
str
Event Triggers
See the full list of default event triggers
rx.select.content
The component that pops out when the select is open.
Prop
Type | Values
Default
Interactive
variant
"solid" | "soft"
color_scheme
"tomato" | "red" | ...
tomato
high_contrast
bool
false
position
"item-aligned" | "popper"
side
"top" | "right" | ...
side_offset
int
align
"start" | "center" | ...
align_offset
```

Event Triggers
See the full list of default event triggers
Trigger
Description
on_close_auto_focus
Fired when the select content is closed.
on_escape_key_down
Fired when the escape key is pressed.
on_pointer_down_outside
Fired when a pointer down event happens outside the select content.
rx.select.group
Used to group multiple items.
Props
No component specific props
Event Triggers
See the full list of default event triggers
rx.select.item
The component that contains the select items.
Prop
Type Values
Default
Interactive
value
str
disabled
bool
false
Event Triggers
See the full list of default event triggers
rx.select.label
Used to render the label of a group, it isn't focusable using arrow keys.
Props
No component specific props

int

Event Triggers

See the full list of default event triggers

rx.select.separator

Used to visually separate items in the Select.

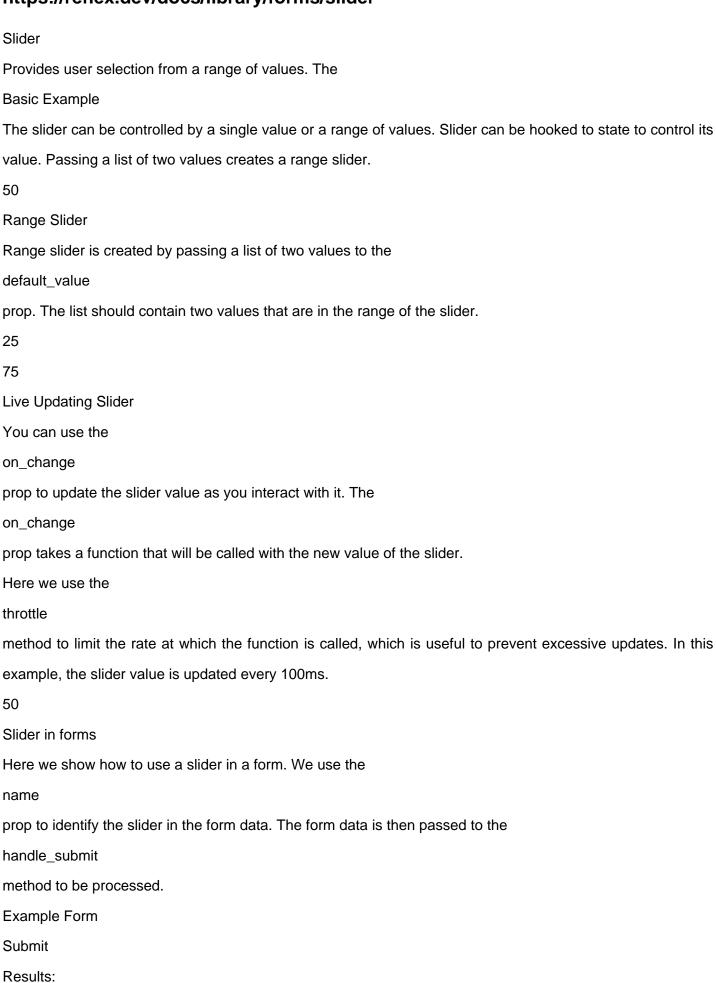
Props

No component specific props

Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/forms/slider

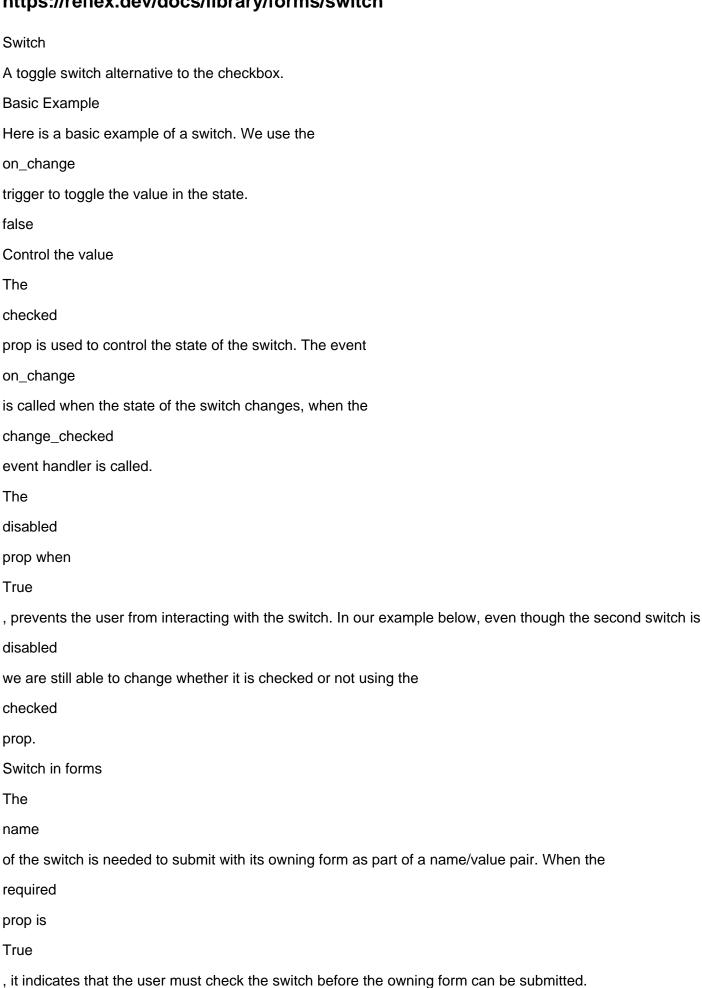


Union[int, float]

max

Union[int, float]
step
Union[int, float]
disabled
bool
false
orientation
"vertical" "horizontal"
Event Triggers
See the full list of default event triggers
Trigger
Description
on_change
Props to rename Fired when the value of the slider changes.
on_value_commit
Fired when a thumb is released after being dragged.

https://reflex.dev/docs/library/forms/switch

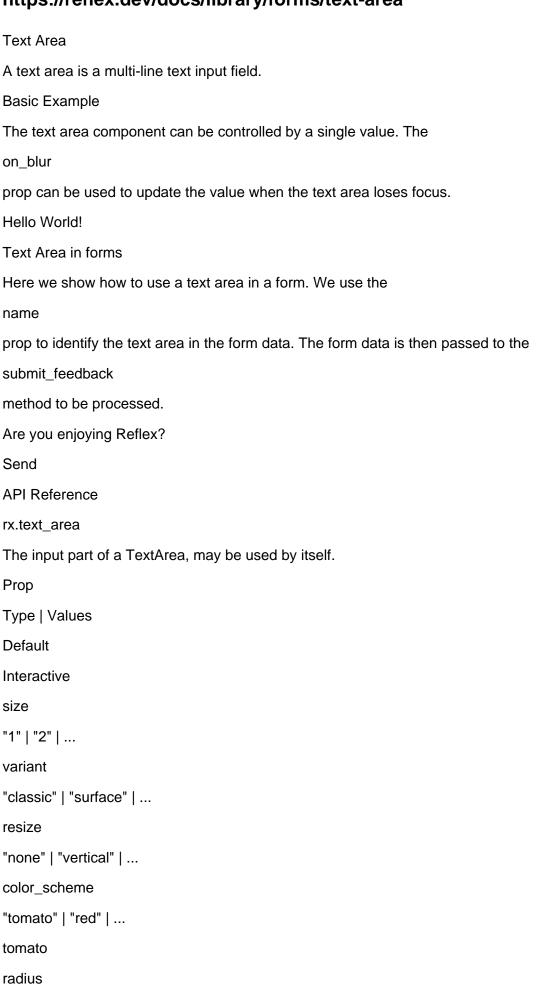


The
value
prop is only used for form submission, use the
checked
prop to control state of the
switch
Example Form
Submit
Results:
{}
API Reference
rx.switch
A toggle switch alternative to the checkbox.
Prop
Type Values
Default
Interactive
as_child
bool
default_checked
bool
checked
bool
disabled
bool
false
required
bool
false
name
str
value

str

```
size
"1" | "2" | ...
variant
"classic" | "surface" | ...
color_scheme
"tomato" | "red" | ...
tomato
high_contrast
bool
false
radius
"none" | "small" | ...
Event Triggers
See the full list of default event triggers
Trigger
Description
on_change
Props to rename Fired when the value of the switch changes
```

https://reflex.dev/docs/library/forms/text-area



"none" "small"
auto_complete
bool
false
auto_focus
bool
false
default_value
str
dirname
str
disabled
bool
false
form
str
max_length
int
min_length
int
name
str
placeholder
str
read_only
bool
false
required
bool
false
rows
str
value
str

wrap

str

Event Triggers

See the full list of default event triggers

Trigger

Description

on_focus

Function or event handler called when the element (or some element inside of it) receives focus. For example, it is called when the user clicks on a text input.

on_blur

Function or event handler called when focus has left the element (or left some element inside of it). For example, it is called when the user clicks outside of a focused text input.

on_change

Function or event handler called when the value of an element has changed. For example, it is called when the user types into a text input each keystroke triggers the on change.

on_key_down

The on_key_down event handler is called when the user presses a key.

on_key_up

The on_key_up event handler is called when the user releases a key.

https://reflex.dev/docs/library/forms/upload

File Upload

Reflex makes it simple to add file upload functionality to your app. You can let users select files, store them on your server, and display or process them as needed. Below is a minimal example that demonstrates how to upload files, save them to disk, and display uploaded images using application state.

Basic File Upload Example

You can let users upload files and keep track of them in your app's state. The example below allows users to upload files, saves them using the backend, and then displays the uploaded files as images.

How File Upload Works

Selecting a file will add it to the browser file list, which can be rendered

on the frontend using the

rx.selected_files(id)

special Var. To clear the

selected files, you can use another special Var

rx.clear_selected_files(id)

as

an event handler.

To upload the file(s), you need to bind an event handler and pass the special

rx.upload_files(upload_id=id)

event arg to it.

File Storage Functions

Reflex provides two key functions for handling uploaded files:

rx.get_upload_dir()

Purpose

: Returns a

Path

object pointing to the server-side directory where uploaded files should be saved

Usage

: Used in backend event handlers to determine where to save uploaded files

Default Location

:

./uploaded_files

(can be customized via

REFLEX_UPLOADED_FILES_DIR environment variable) Type : Returns pathlib.Path rx.get_upload_url(filename) **Purpose** : Returns the URL string that can be used in frontend components to access uploaded files Usage : Used in frontend components (like rx.image rx.video) to display uploaded files **URL Format** /_upload/filename Type : Returns str **Key Differences** rx.get_upload_dir() -> Backend file path for saving files rx.get_upload_url() -> Frontend URL for displaying files **Basic Upload Pattern** Here is the standard pattern for handling file uploads: Multiple File Upload Below is an example of how to allow multiple file uploads (in this case images). Uploading a Single File (Video) Below is an example of how to allow only a single file upload and render (in this case a video). Customizing the Upload

In the example below, the upload component accepts a maximum number of 5 files of specific types.

It also disables the use of the space or enter key in uploading files.

```
To use a one-step upload, bind the event handler to the
rx.upload
component's
on_drop
trigger.
Unstyled Upload Component
To use a completely unstyled upload component and apply your own customization, use
rx.upload.root
instead:
Click to upload
or drag and drop
SVG, PNG, JPG or GIF (MAX. 5MB)
Handling the Upload
Your event handler should be an async function that accepts a single argument,
files: list[UploadFile]
, which will contain
FastAPI UploadFile
instances.
You can read the files and save them anywhere as shown in the example.
In your UI, you can bind the event handler to a trigger, such as a button
on_click
event or upload
on_drop
event, and pass in the files using
rx.upload_files()
Saving the File
By convention, Reflex provides the function
rx.get_upload_dir()
to get the directory where uploaded files may be saved. The upload dir comes from the environment variable
REFLEX_UPLOADED_FILES_DIR
, or
./uploaded_files
if not specified.
```

The backend of your app will mount this uploaded files directory on /_upload without restriction. Any files uploaded via this mechanism will automatically be publicly accessible. To get the URL for a file inside the upload dir, use the rx.get_upload_url(filename) function in a frontend component. When using the Reflex hosting service, the uploaded files directory is not persistent and will be cleared on every deployment. For persistent storage of uploaded files, it is recommended to use an external service, such as S3. **Directory Structure and URLs** By default, Reflex creates the following structure: The files are automatically served at: /_upload/image1.png ↕ rx.get_upload_url("image1.png") / upload/document.pdf ↕ rx.get_upload_url("document.pdf") /_upload/video.mp4 ↕ rx.get_upload_url("video.mp4") Cancellation The id provided to the rx.upload component can be passed to the special event handler rx.cancel_upload(id) to stop uploading on demand. Cancellation can be triggered directly by a frontend event trigger, or it can be returned from a backend event handler. **Progress** The rx.upload_files

special event arg also accepts an

on_upload_progress event trigger which will be fired about every second during the upload operation to report the progress of the upload. This can be used to update a progress bar or other UI elements to show the user the progress of the upload. The progress dictionary contains the following keys: **API Reference** rx.upload The styled Upload Component. Prop Type | Values Default accept Union[dict, NoneType] disabled bool max_files int max_size int min_size int multiple bool no_click bool no_drag bool no_keyboard bool **Event Triggers** See the full list of default event triggers

Trigger

Description
on_drop
Marked True when any Upload component is created. Fired when files are dropped.
rx.upload.root
A file upload component.
Prop
Type Values
Default
accept
Union[dict, NoneType]
disabled
bool
max_files
int
max_size
int
min_size
int
multiple
bool
no_click
bool
no_drag
bool
no_keyboard
bool
Event Triggers
See the full list of default event triggers
Trigger
Description
on_drop
Marked True when any Upload component is created. Fired when files are dropped.

https://reflex.dev/docs/library/graphing/charts

Charts Components for creating various types of charts and graphs. These are useful for data visualization and presenting complex information in an easily understandable format. Areachart Barchart Composedchart Errorbar

Funnelchart

Linechart

Piechart

Radarchart

Radialbarchart

Scatterchart

https://reflex.dev/docs/library/graphing/charts/areachart

Area Chart
A Recharts area chart displays quantitative data using filled areas between a line connecting data points and
the axis.
Basic Example
Code
Code
Data
Data
Syncing Charts
The
sync_id
prop allows you to sync two graphs. In the example, it is set to "1" for both charts, indicating that they should
be synchronized. This means that any interactions (such as brushing) performed on one chart will be
reflected in the other chart.
Code
Code
Data
Data
Stacking Charts
The
stack_id
prop allows you to stack multiple graphs on top of each other. In the example, it is set to "1" for both charts
indicating that they should be stacked together. This means that the bars or areas of the charts will be
vertically stacked, with the values of each chart contributing to the total height of the stacked areas or bars.
This is similar to the
sync_id
prop, but instead of synchronizing the interaction between the charts, it just stacks the charts on top of each
other.
Code
Code
Data
Data

Multiple Axis
Multiple axes can be used for displaying different data series with varying scales or units on the same chart.
This allows for a more comprehensive comparison and analysis of the data.
Code
Code
Data
Data
Layout
Use the
layout
prop to set the orientation to either
"horizontal"
(default) or
"vertical"
•
Include margins around your graph to ensure proper spacing and enhance readability. By default, provide
margins on all sides of the chart to create a visually appealing and functional representation of your data.
Code
Code
Data
Data
Stateful Example
Here is an example of an area graph with a
State
. Here we have defined a function
randomize_data
, which randomly changes the data for both graphs when the first defined
area
is clicked on using
on_click=AreaState.randomize_data
•
Curve Type:
API Reference
rx.recharts.AreaChart

An Area chart component in Recharts.
Prop
Type Values
Default
base_value
"dataMin" "dataMax"
"auto"
data
Sequence
margin
Dict[str, Any]
sync_id
str
sync_method
"index" "value"
"index"
layout
"vertical" "horizontal"
"horizontal"
stack_offset
"expand" "none"
width
Union[str, int]
Var.create("100%")
height
Union[str, int]
Var.create("100%")
Valid Children
XAxis
YAxis
ReferenceArea
ReferenceDot
ReferenceLine
Brush

CartesianGrid
Legend
GraphingTooltip
Area
Defs
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the component in this chart
rx.recharts.Area
An Area component in Recharts.
Prop
Type Values
Default
stroke
Union[str, Color]
rx.color("accent", 9)
stroke_width
Union[str, int, float]
1
fill
Union[str, Color]
rx.color("accent", 5)
type_
"basis" "basisClosed"
"monotone"
dot
Union[dict, bool]
False
active_dot
Union[dict, bool]
{stroke: rx.color("accent", 2), fill: rx.color("accent", 10)}

```
base_line
Union[int, Sequence]
points
Sequence
stack_id
Union[str, int]
connect_nulls
bool
False
layout
"vertical" | "horizontal"
data_key
Union[str, int]
x_axis_id
Union[str, int]
0
y_axis_id
Union[str, int]
0
legend_type
"circle" | "cross" | ...
label
Union[dict, bool]
False
is_animation_active
bool
True
animation_begin
int
0
animation_duration
int
1500
animation_easing
```

"ease" "ease-in"
"ease"
unit
Union[str, int]
name
Union[str, int]
Valid Children
LabelList
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the component in this group
on_animation_start
The customized event handler of animation start
on_animation_end
The customized event handler of animation end

https://reflex.dev/docs/library/graphing/charts/barchart

Bar Chart A bar chart presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. For a bar chart we must define an rx.recharts.bar() component for each set of values we wish to plot. Each rx.recharts.bar() component has a data_key which clearly states which variable in our data we are tracking. In this simple example we plot uν as a bar against the name column which we set as the data_key in rx.recharts.x_axis Simple Example Code Code Data Data Multiple Bars Multiple bars can be placed on the same bar_chart , using multiple rx.recharts.bar() components. Code Code

Data

Data
Ranged Charts
You can also assign a range in the bar by assigning the data_key in the
rx.recharts.bar
to a list with two elements, i.e. here a range of two temperatures for each date.
Code
Code
Data
Data
Stateful Charts
Here is an example of a bar graph with a
State
. Here we have defined a function
randomize_data
, which randomly changes the data for both graphs when the first defined
bar
is clicked on using
on_click=BarState.randomize_data
Example with Props
Here's an example demonstrates how to customize the appearance and layout of bars using the
bar_category_gap
,
bar_gap
,
bar_size
, and
max_bar_size
props. These props accept values in pixels to control the spacing and size of the bars.
Code
Code
Data
Data
Vertical Example

The
layout
prop allows you to set the orientation of the graph to be vertical or horizontal, it is set horizontally by default.
Include margins around your graph to ensure proper spacing and enhance readability. By default, provide
margins on all sides of the chart to create a visually appealing and functional representation of your data.
Code
Code
Data
Data
To learn how to use the
sync_id
,
stack_id
,
x_axis_id
and
y_axis_id
props check out the of the area chart
documentation
, where these props are all described with examples.
API Reference
rx.recharts.BarChart
A Bar chart component in Recharts.
Prop
Type Values
Default
bar_category_gap
Union[str, int]
"10%"
bar_gap
Union[str, int]
4
bar_size
int

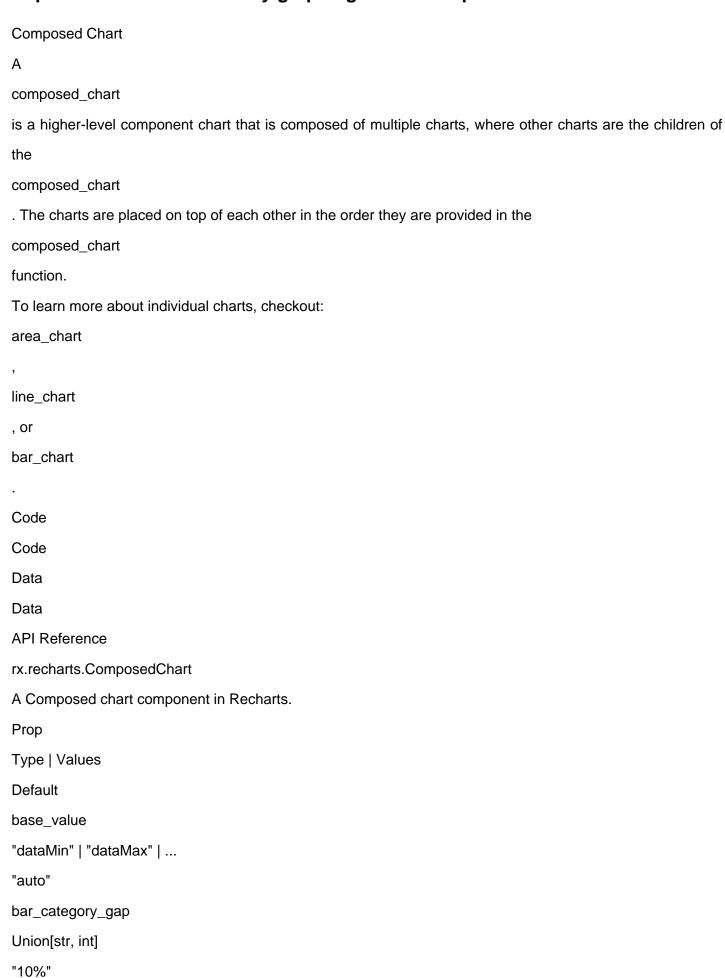
max_bar_size
int
stack_offset
"expand" "none"
"none"
reverse_stack_order
bool
False
data
Sequence
margin
Dict[str, Any]
sync_id
str
sync_method
"index" "value"
"index"
layout
"vertical" "horizontal"
"horizontal"
width
Union[str, int]
Var.create("100%")
height
Union[str, int]
Var.create("100%")
Valid Children
XAxis
YAxis
ReferenceArea
ReferenceDot
ReferenceLine
Brush
CartesianGrid

Legend
GraphingTooltip
Bar
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the component in this chart
rx.recharts.Bar
A Bar component in Recharts.
Prop
Type Values
Default
stroke
Union[str, Color]
stroke_width
Union[str, int, float]
fill
Union[str, Color]
Color("accent", 9)
background
bool
False
stack_id
str
unit
Union[str, int]
min_point_size
int
name
Union[str, int]
bar_size
int

```
max_bar_size
int
radius
Union[int, Sequence]
layout
"vertical" | "horizontal"
data_key
Union[str, int]
x_axis_id
Union[str, int]
0
y_axis_id
Union[str, int]
0
legend_type
"circle" | "cross" | ...
label
Union[dict, bool]
False
is_animation_active
bool
True
animation_begin
int
0
animation_duration
int
1500
animation_easing
"ease" | "ease-in" | ...
"ease"
Valid Children
Cell
```

LabelList
ErrorBar
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the component in this group
on_animation_start
The customized event handler of animation start
on_animation_end
The customized event handler of animation end

https://reflex.dev/docs/library/graphing/charts/composedchart



bar_gap
int
4
bar_size
int
reverse_stack_order
bool
False
data
Sequence
margin
Dict[str, Any]
sync_id
str
sync_method
"index" "value"
"index"
layout
"vertical" "horizontal"
"horizontal"
stack_offset
"expand" "none"
width
Union[str, int]
Var.create("100%")
height
Union[str, int]
Var.create("100%")
Valid Children
XAxis
YAxis
ReferenceArea
ReferenceDot
ReferenceLine

https://reflex.dev/docs/library/graphing/charts/errorbar



Union[str, Color]

rx.color("gray", 8)

stroke_width

Union[str, int, float]

1.5

Event Triggers

See the full list of default event triggers

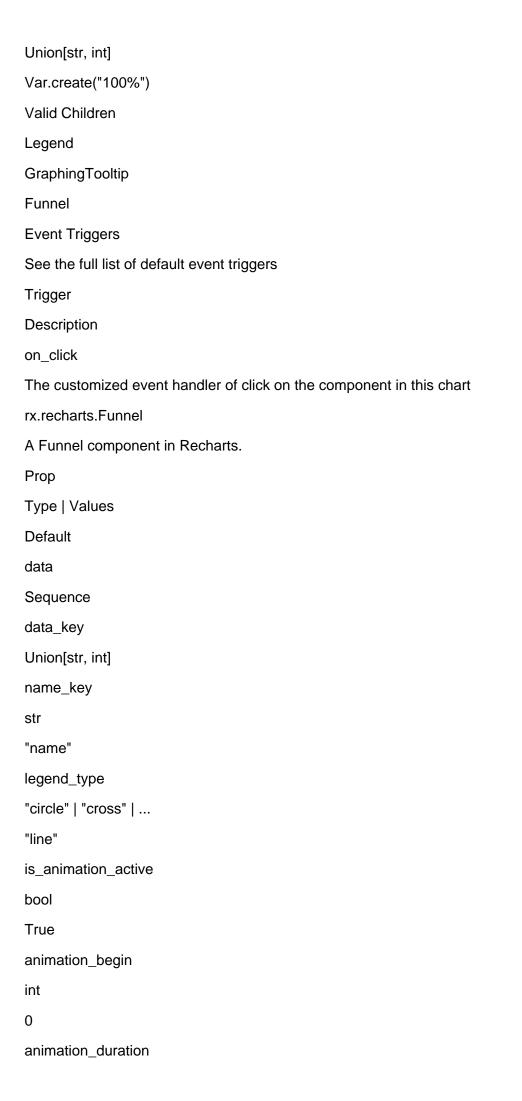
https://reflex.dev/docs/library/graphing/charts/funnelchart

Funnel Chart

A funnel chart is a graphical representation used to visualize how data moves through a process. In a funnel the dependent variable's value diminishes in the subsequent sta

chart, the dependent variable's value diminishes in the subsequent stages of the process. It can be used
to demonstrate the flow of users through a business or sales process.
Simple Example
Code
Code
Data
Data
Event Triggers
Funnel chart supports
on_click
,
on_mouse_enter
,
on_mouse_leave
and
on_mouse_move
event triggers, allows you to interact with the funnel chart and perform specific actions based on user
interactions.
Code
Code
Data
Data
Dynamic Data
Here is an example of a funnel chart with a
State
. Here we have defined a function
randomize_data
, which randomly changes the data when the graph is clicked on using
on_click=FunnelState.randomize_data

Changing the Chart Animation The is_animation_active prop can be used to turn off the animation, but defaults to True animation_begin sets the delay before animation starts, animation_duration determines how long the animation lasts, and animation_easing defines the speed curve of the animation for smooth transitions. Code Code Data Data **API Reference** rx.recharts.FunnelChart A Funnel chart component in Recharts. Prop Type | Values Default layout str "centric" margin Dict[str, Any] {"top": 5, "right": 5, "bottom": 5, "left": 5} stroke Union[str, Color] width Union[str, int] Var.create("100%") height

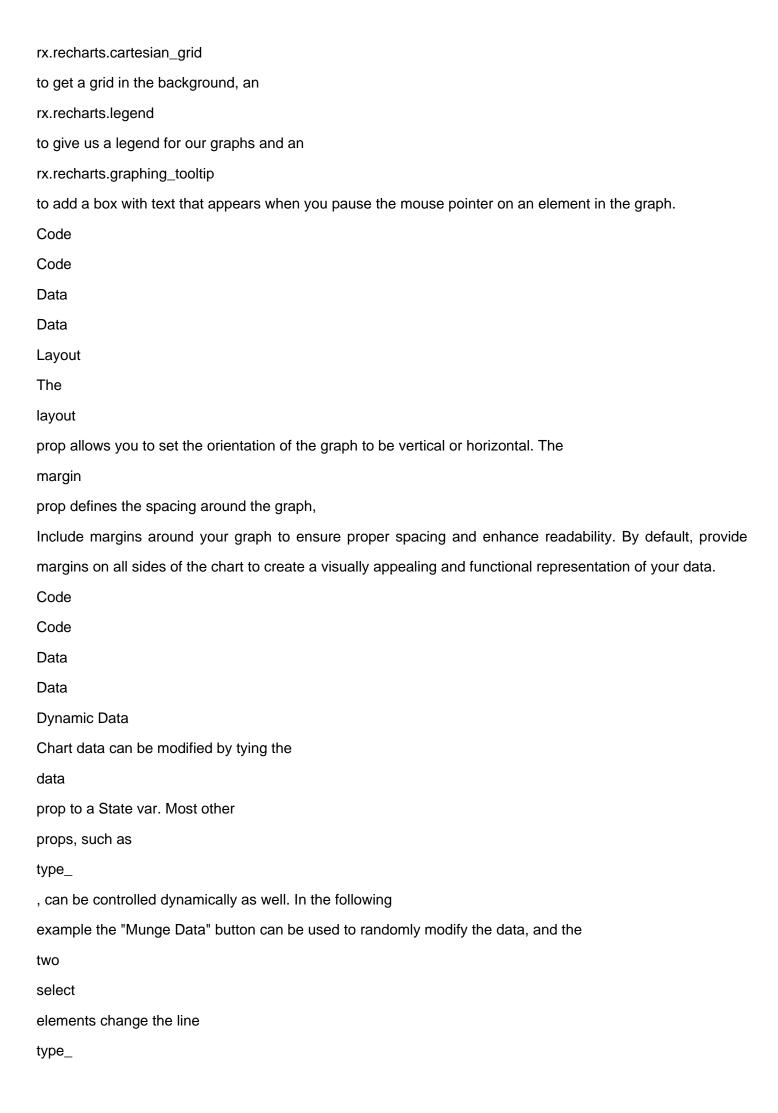


```
int
1500
animation_easing
"ease" | "ease-in" | ...
stroke
Union[str, Color]
rx.color("gray", 3)
trapezoids
Sequence
Valid Children
LabelList
Cell
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the component in this group
on_animation_start
Valid children components The customized event handler of animation start
on_animation_end
The customized event handler of animation end
```

https://reflex.dev/docs/library/graphing/charts/linechart

to style the lines differently. In addition, we add an

Line Chart A line chart is a type of chart used to show information that changes over time. Line charts are created by plotting a series of several points and connecting them with a straight line. Simple Example For a line chart we must define an rx.recharts.line() component for each set of values we wish to plot. Each rx.recharts.line() component has a data_key which clearly states which variable in our data we are tracking. In this simple example we plot рν and uν as separate lines against the name column which we set as the data_key in rx.recharts.x_axis Code Code Data Data Example with Props Our second example uses exactly the same data as our first example, except now we add some extra features to our line graphs. We add a type_ prop to rx.recharts.line



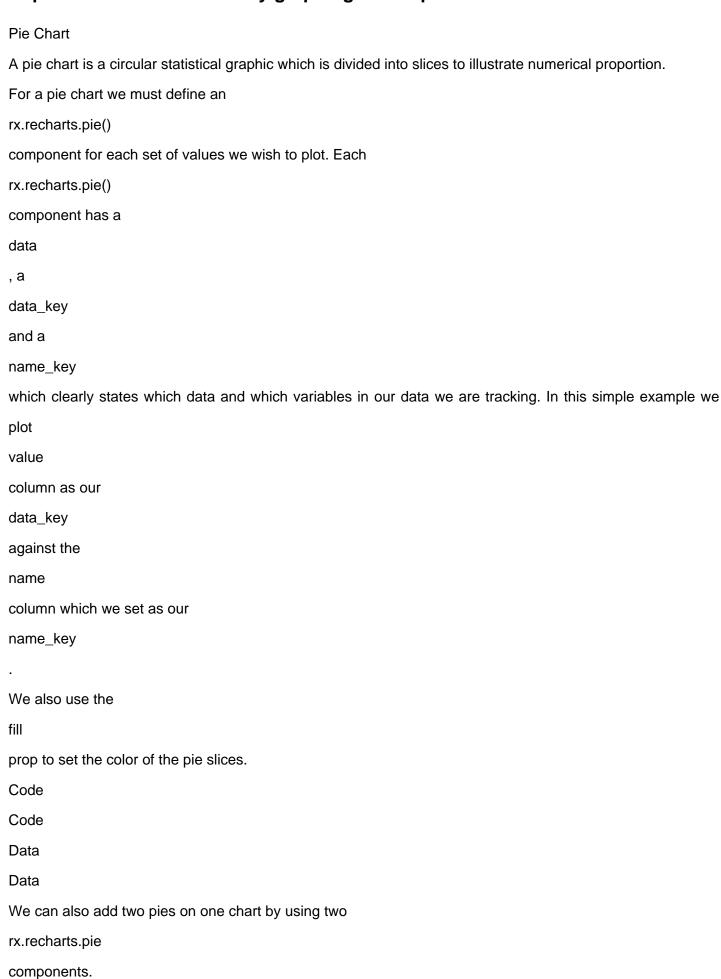
. Since the data and style is saved in the per-browser-tab State, the changes will not be visible to other visitors. Munge Data To learn how to use the sync_id x_axis_id and y_axis_id props check out the of the area chart documentation , where these props are all described with examples. **API Reference** rx.recharts.LineChart A Line chart component in Recharts. Prop Type | Values Default data Sequence margin Dict[str, Any] sync_id str sync_method "index" | "value" "index" layout "vertical" | "horizontal" "horizontal" stack_offset "expand" | "none" | ... width Union[str, int]

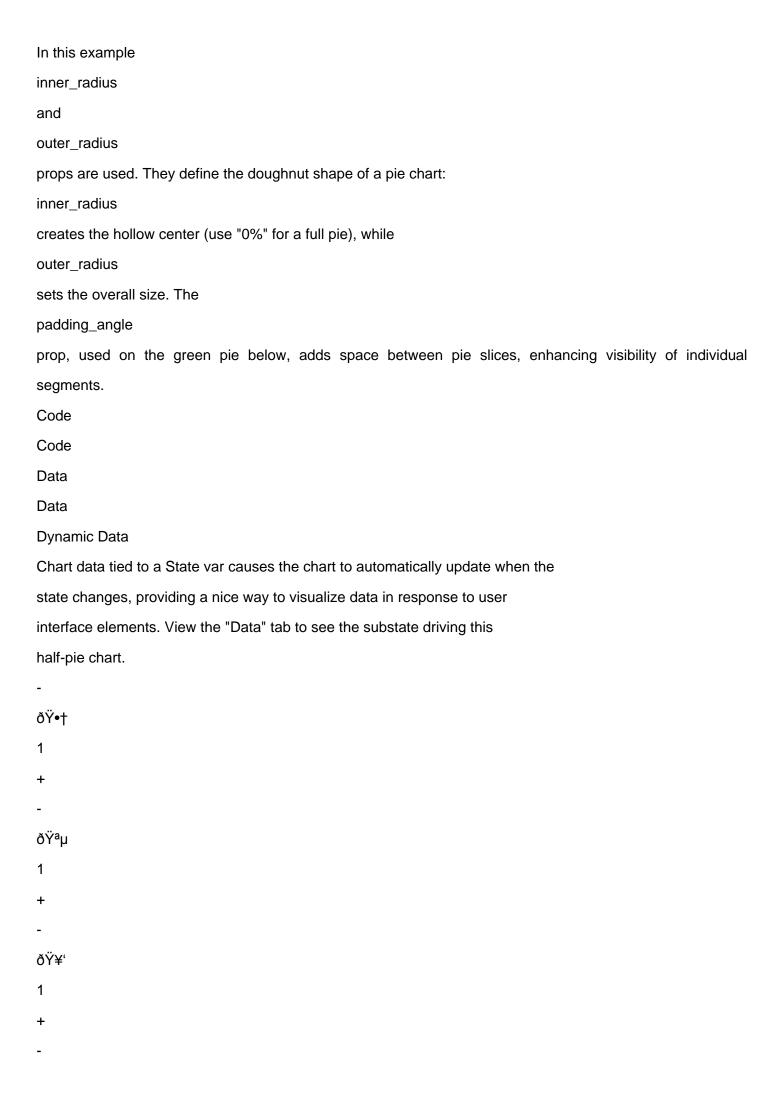
Var.create("100%")
height
Union[str, int]
Var.create("100%")
Valid Children
XAxis
YAxis
ReferenceArea
ReferenceDot
ReferenceLine
Brush
CartesianGrid
Legend
GraphingTooltip
Line
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the component in this chart
rx.recharts.Line
A Line component in Recharts.
Prop
Type Values
Default
type_
"basis" "basisClosed"
stroke
Union[str, Color]
rx.color("accent", 9)
stroke_width
Union[str, int, float]

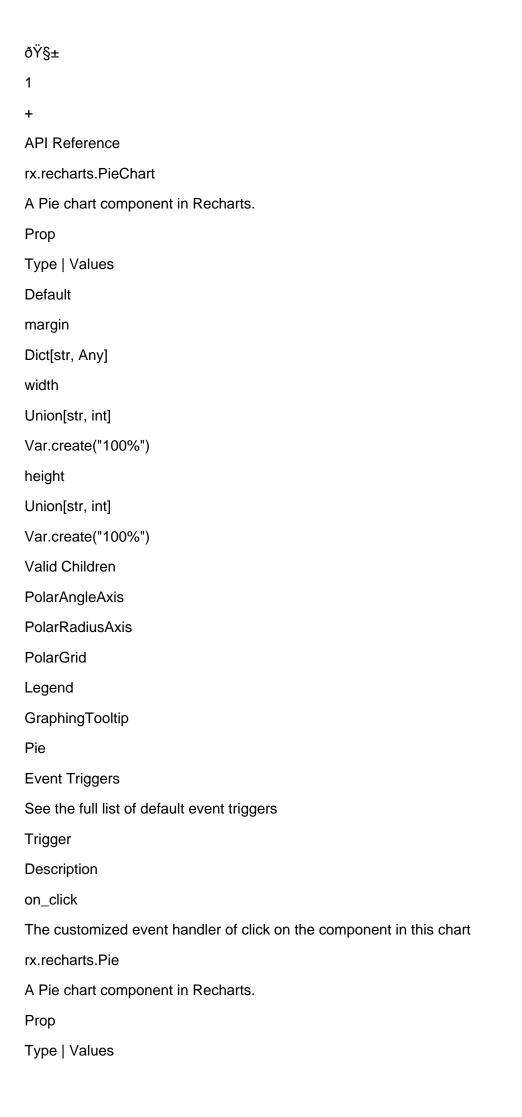
```
dot
Union[dict, bool]
{"stroke": rx.color("accent", 10), "fill": rx.color("accent", 4)}
active_dot
Union[dict, bool]
{"stroke": rx.color("accent", 2), "fill": rx.color("accent", 10)}
hide
bool
False
connect_nulls
bool
unit
Union[str, int]
points
Sequence
stroke_dasharray
str
layout
"vertical" | "horizontal"
data_key
Union[str, int]
x_axis_id
Union[str, int]
0
y_axis_id
Union[str, int]
0
legend_type
"circle" | "cross" | ...
label
Union[dict, bool]
False
is_animation_active
bool
```

True
animation_begin
int
0
animation_duration
int
1500
animation_easing
"ease" "ease-in"
"ease"
name
Union[str, int]
Valid Children
LabelList
ErrorBar
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the component in this group
on_animation_start
The customized event handler of animation start
on_animation_end
The customized event handler of animation end

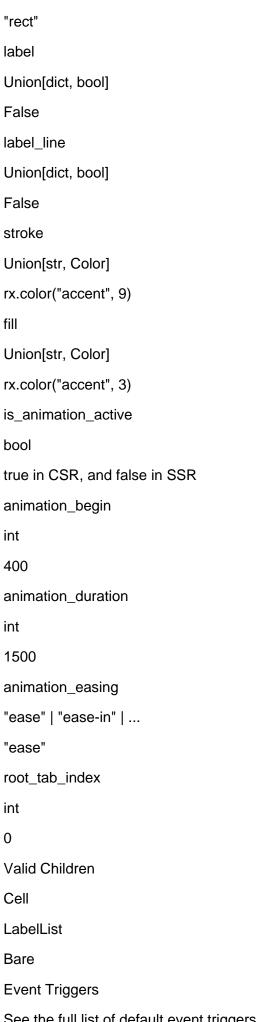
https://reflex.dev/docs/library/graphing/charts/piechart







Default
data
Sequence
data_key
Union[str, int]
сх
Union[str, int]
"50%"
су
Union[str, int]
"50%"
inner_radius
Union[str, int]
0
outer_radius
Union[str, int]
"80%"
start_angle
int
0
end_angle
int
360
min_angle
int
0
padding_angle
int
0
name_key
str
"name"
legend_type
"circle" "cross"



See the full list of default event triggers

Trigger

Description

on_animation_start

The on_animation_start event handler is called when the animation starts. It receives the animation name as an argument.

on_animation_end

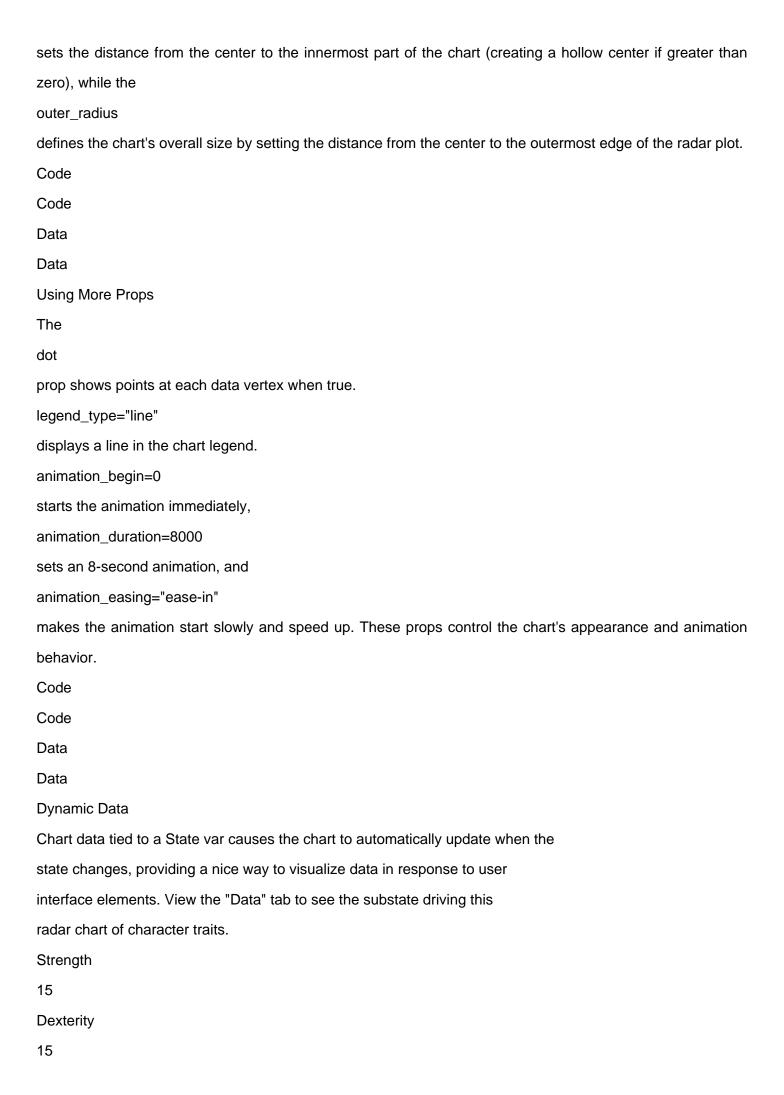
The on_animation_end event handler is called when the animation ends. It receives the animation name as an argument.

on_click

Function or event handler called when the user clicks on an element. For example, it's called when the user clicks on a button.

https://reflex.dev/docs/library/graphing/charts/radarchart

Radar Chart A radar chart shows multivariate data of three or more quantitative variables mapped onto an axis. Simple Example For a radar chart we must define an rx.recharts.radar() component for each set of values we wish to plot. Each rx.recharts.radar() component has a data_key which clearly states which variable in our data we are plotting. In this simple example we plot the Α column of our data against the subject column which we set as the data_key in rx.recharts.polar_angle_axis Code Code Data Data Multiple Radars We can also add two radars on one chart by using two rx.recharts.radar components. In this plot an inner_radius and an outer_radius are set which determine the chart's size and shape. The inner_radius



inner_radius

Union[str, int]
0
outer_radius
Union[str, int]
"80%"
width
Union[str, int]
Var.create("100%")
height
Union[str, int]
Var.create("100%")
Valid Children
PolarAngleAxis
PolarRadiusAxis
PolarGrid
Legend
GraphingTooltip
Radar
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the component in this chart
rx.recharts.Radar
A Radar chart component in Recharts.
Prop
Type Values
Default
data_key
Union[str, int]
points
Sequence
dot

```
Union[dict, bool]
True
stroke
Union[str, Color]
rx.color("accent", 9)
fill
Union[str, Color]
rx.color("accent", 3)
fill_opacity
float
0.6
legend_type
"circle" | "cross" | ...
"rect"
label
Union[dict, bool]
True
is_animation_active
bool
True in CSR, and False in SSR
animation_begin
int
0
animation_duration
int
1500
animation_easing
"ease" | "ease-in" | ...
"ease"
Valid Children
LabelList
Event Triggers
See the full list of default event triggers
Trigger
```

Description

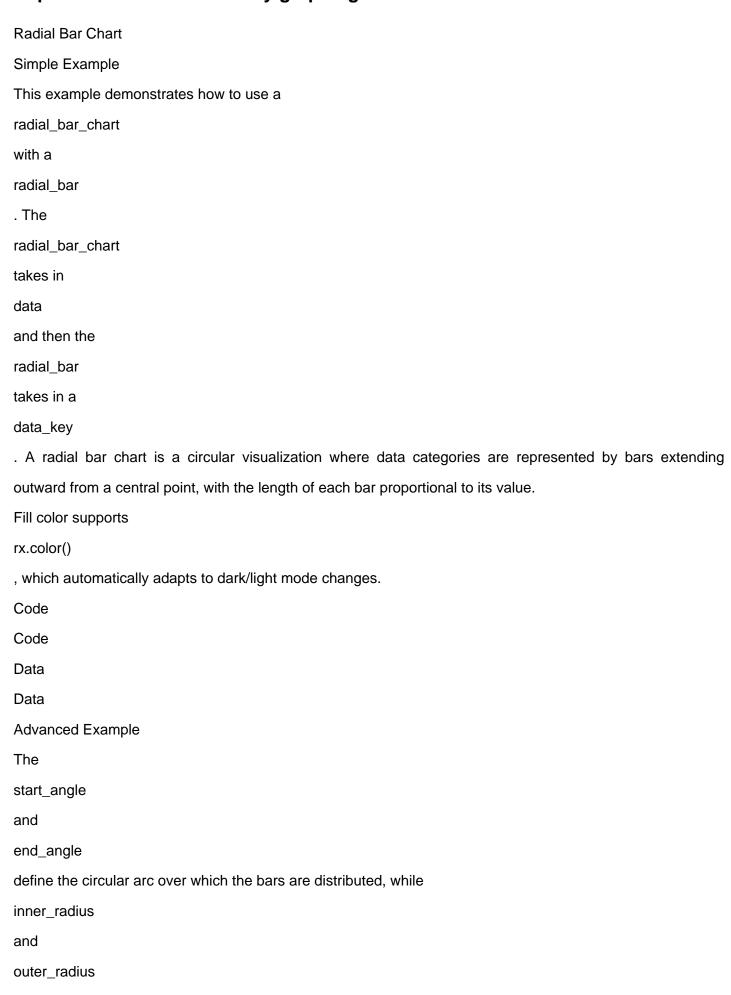
on_animation_start

The on_animation_start event handler is called when the animation starts. It receives the animation name as an argument.

on_animation_end

The on_animation_end event handler is called when the animation ends. It receives the animation name as an argument.

https://reflex.dev/docs/library/graphing/charts/radialbarchart



determine the radial extent of the bars from the center.
Code
Code
Data
Data
API Reference
rx.recharts.RadialBarChart
A RadialBar chart component in Recharts.
Prop
Type Values
Default
data
Sequence
margin
Dict[str, Any]
{"top": 5, "right": 5, "left": 5 "bottom": 5}
СХ
Union[str, int]
"50%"
су
Union[str, int]
"50%"
start_angle
int
0
end_angle
int
360
inner_radius
Union[str, int]
"30%"
outer_radius
Union[str, int]
"100%"

bar_category_gap
Union[str, int]
"10%"
bar_gap
str
4
bar_size
int
width
Union[str, int]
Var.create("100%")
height
Union[str, int]
Var.create("100%")
Valid Children
PolarAngleAxis
PolarRadiusAxis
PolarGrid
Legend
GraphingTooltip
RadialBar
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the component in this chart

https://reflex.dev/docs/library/graphing/charts/scatterchart

Scatter Chart

To learn how to use the

x_axis_id

A scatter chart always has two value axes to show one set of numerical data along a horizontal (value) axis ersection

and another set of numerical values along a vertical (value) axis. The chart displays points at the interest of numerical values along a vertical (value) axis.
of an x and y numerical value, combining these values into single data points.
Simple Example
For a scatter chart we must define an
rx.recharts.scatter()
component for each set of values we wish to plot. Each
rx.recharts.scatter()
component has a
data
prop which clearly states which data source we plot. We also must define
rx.recharts.x_axis()
and
rx.recharts.y_axis()
so that the graph knows what data to plot on each axis.
Code
Code
Data
Data
Multiple Scatters
We can also add two scatters on one chart by using two
rx.recharts.scatter()
components, and we can define an
rx.recharts.z_axis()
which represents a third column of data and is represented by the size of the dots in the scatter plot.
Code
Code
Data
Data

```
and
y_axis_id
props, check out the Multiple Axis section of the area chart
documentation
Dynamic Data
Chart data tied to a State var causes the chart to automatically update when the
state changes, providing a nice way to visualize data in response to user
interface elements. View the "Data" tab to see the substate driving this
calculation of iterations in the Collatz Conjecture for a given starting number.
Enter a starting number in the box below the chart to recalculate.
Compute
Legend Type and Shape
Legend Type:
Shape:
API Reference
rx.recharts.ScatterChart
A Scatter chart component in Recharts.
Prop
Type | Values
Default
margin
Dict[str, Any]
{"top": 5, "right": 5, "bottom": 5, "left": 5}
width
Union[str, int]
Var.create("100%")
height
Union[str, int]
Var.create("100%")
```

Valid Children

XAxis

YAxis

ZAxis

ReferenceArea
ReferenceDot
ReferenceLine
Brush
CartesianGrid
Legend
GraphingTooltip
Scatter
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the component in this chart
rx.recharts.Scatter
A Scatter component in Recharts.
Prop
Type Values
Default
data
Sequence
name
str
legend_type
"circle" "cross"
"circle"
x_axis_id
Union[str, int]
0
y_axis_id
Union[str, int]
0
z_axis_id
Union[str, int]

```
line
bool
False
shape
"square" | "circle" | ...
"circle"
line_type
"joint" | "fitting"
"joint"
fill
Union[str, Color]
rx.color("accent", 9)
is_animation_active
bool
True in CSR, False in SSR
animation_begin
int
0
animation_duration
int
1500
animation_easing
"ease" | "ease-in" | ...
"ease"
Valid Children
LabelList
ErrorBar
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the component in this group
```

0

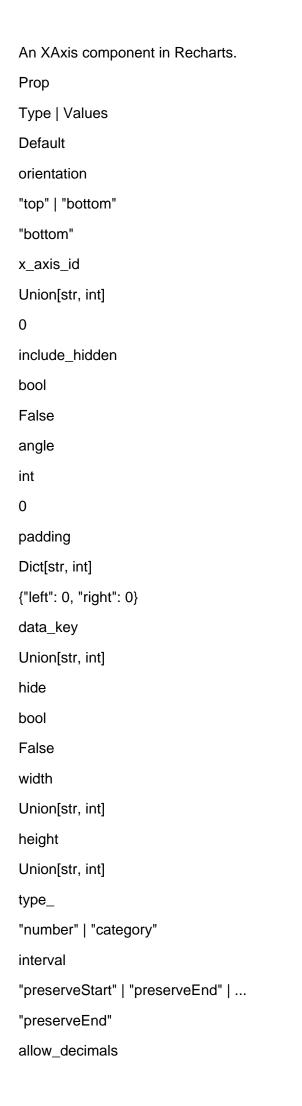
https://reflex.dev/docs/library/graphing/general

General
General-purpose graphing components that provide foundational elements for creating custom visualizations
These components offer flexibility and can be combined to create more complex graphical representations.
Axis
Brush
Cartesiangrid
Label
Legend
Reference
Tooltip

https://reflex.dev/docs/library/graphing/general/axis

rx.recharts.XAxis

Axis The Axis component in Recharts is a powerful tool for customizing and configuring the axes of your charts. It provides a wide range of props that allow you to control the appearance, behavior, and formatting of the axis. Whether you're working with an AreaChart, LineChart, or any other chart type, the Axis component enables you to create precise and informative visualizations. **Basic Example** Code Code Data Data Multiple Axes Multiple axes can be used for displaying different data series with varying scales or units on the same chart. This allows for a more comprehensive comparison and analysis of the data. Code Code Data Data Choosing Location of Labels for Axes The axes label can take several positions. The example below allows you to try out different locations for the x and y axis labels. X Label Position: X Label Offset: Y Label Position: Y Label Offset: Code Code Data Data **API** Reference



bool
True
allow_data_overflow
bool
False
allow_duplicated_category
bool
True
domain
Sequence
[0, "auto"]
axis_line
bool
True
mirror
bool
False
reversed
bool
False
label
Union[str, int, dict]
scale
"auto" "linear"
"auto"
unit
Union[str, int]
name
Union[str, int]
ticks
Sequence
tick
Union[bool, dict]
tick_count

```
int
5
tick_line
bool
True
tick_size
int
6
min_tick_gap
int
5
stroke
Union[str, Color]
rx.color("gray", 9)
text_anchor
"start" | "middle" | ...
"middle"
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the ticks of this axis
rx.recharts.YAxis
A YAxis component in Recharts.
Prop
Type | Values
Default
orientation
"left" | "right"
"left"
y_axis_id
Union[str, int]
```

```
padding
Dict[str, int]
{"top": 0, "bottom": 0}
data_key
Union[str, int]
hide
bool
False
width
Union[str, int]
height
Union[str, int]
type_
"number" | "category"
interval
"preserveStart" | "preserveEnd" | ...
"preserveEnd"
allow_decimals
bool
True
allow_data_overflow
bool
False
allow_duplicated_category
bool
True
domain
Sequence
[0, "auto"]
axis_line
bool
True
mirror
bool
```

False
reversed
bool
False
label
Union[str, int, dict]
scale
"auto" "linear"
"auto"
unit
Union[str, int]
name
Union[str, int]
ticks
Sequence
tick
Union[bool, dict]
tick_count
int
5
tick_line
bool
True
tick_size
int
6
min_tick_gap
int
5
stroke
Union[str, Color]
rx.color("gray", 9)
text_anchor
"start" "middle"

"middle"
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
The customized event handler of click on the ticks of this axis
rx.recharts.ZAxis
A ZAxis component in Recharts.
Prop
Type Values
Default
data_key
Union[str, int]
z_axis_id
Union[str, int]
0
range
Sequence
[60, 400]
unit
Union[str, int]
name
Union[str, int]
scale
"auto" "linear"
"auto"
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/graphing/general/brush



fill
Union[str, Color]
rx.color("gray", 2)
data_key
Union[str, int]
x
int
0
у
int
0
width
int
0
height
int
40
data
Sequence
traveller_width
int
5
gap
int
1
start_index
int
0
and inday
end_index
int
int
int fill

Event Triggers

See the full list of default event triggers

Trigger

Description

on_change

Function or event handler called when the value of an element has changed. For example, it is called when the user types into a text input each keystroke triggers the on change.

https://reflex.dev/docs/library/graphing/general/cartesiangrid

Cartesian Grid The Cartesian Grid is a component in Recharts that provides a visual reference for data points in charts. It helps users to better interpret the data by adding horizontal and vertical lines across the chart area. Simple Example The stroke_dasharray prop in Recharts is used to create dashed or dotted lines for various chart elements like lines, axes, or grids. It's based on the SVG stroke-dasharray attribute. The stroke_dasharray prop accepts a comma-separated string of numbers that define a repeating pattern of dashes and gaps along the length of the stroke. stroke_dasharray="5,5" : creates a line with 5-pixel dashes and 5-pixel gaps stroke dasharray="10,5,5,5" : creates a more complex pattern with 10-pixel dashes, 5-pixel gaps, 5-pixel dashes, and 5-pixel gaps Here's a simple example using it on a Line component: Code Code Data Data Hidden Axes Α cartesian_grid component can be used to hide the horizontal and vertical grid lines in a chart by setting the horizontal and vertical

. This can be useful when you want to show the grid lines only on one axis or when you want to create a cleaner look for the chart.

Code

props to

False

Code
Data
Data
Custom Grid Lines
The
horizontal_points
and
vertical_points
props allow you to specify custom grid lines on the chart, offering fine-grained control over the grid's
appearance.
These props accept arrays of numbers, where each number represents a pixel offset:
For
horizontal_points
, the offset is measured from the top edge of the chart
For
vertical_points
, the offset is measured from the left edge of the chart
Important
: The values provided to these props are not directly related to the axis values. They represent pixel offsets
within the chart's rendering area.
Here's an example demonstrating custom grid lines in a scatter chart:
Code
Code
Data
Data
Use these props judiciously to enhance data visualization without cluttering the chart. They're particularly
useful for highlighting specific data ranges or creating visual reference points.
API Reference
rx.recharts.CartesianGrid
A CartesianGrid component in Recharts.
Prop
Type Values
Default
horizontal

bool
True
vertical
bool
True
vertical_points
Sequence
horizontal_points
Sequence
fill
Union[str, Color]
fill_opacity
float
stroke_dasharray
str
stroke
Union[str, Color]
rx.color("gray", 7)
x
int
0
у
int
0
width
int
0
height
int
0
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/graphing/general/label

Label Label is a component used to display a single label at a specific position within a chart or axis, while LabelList is a component that automatically renders a list of labels for each data point in a chart series, providing a convenient way to display multiple labels without manually positioning each one. Simple Example Here's a simple example that demonstrates how you can customize the label of your axis using rx.recharts.label . The value prop represents the actual text of the label, the position prop specifies where the label is positioned within the axis component, and the offset prop is used to fine-tune the label's position. Code Code Data Data Label List Example rx.recharts.label_list takes in a data_key where we define the data column to plot. Code Code Data Data **API** Reference rx.recharts.Label

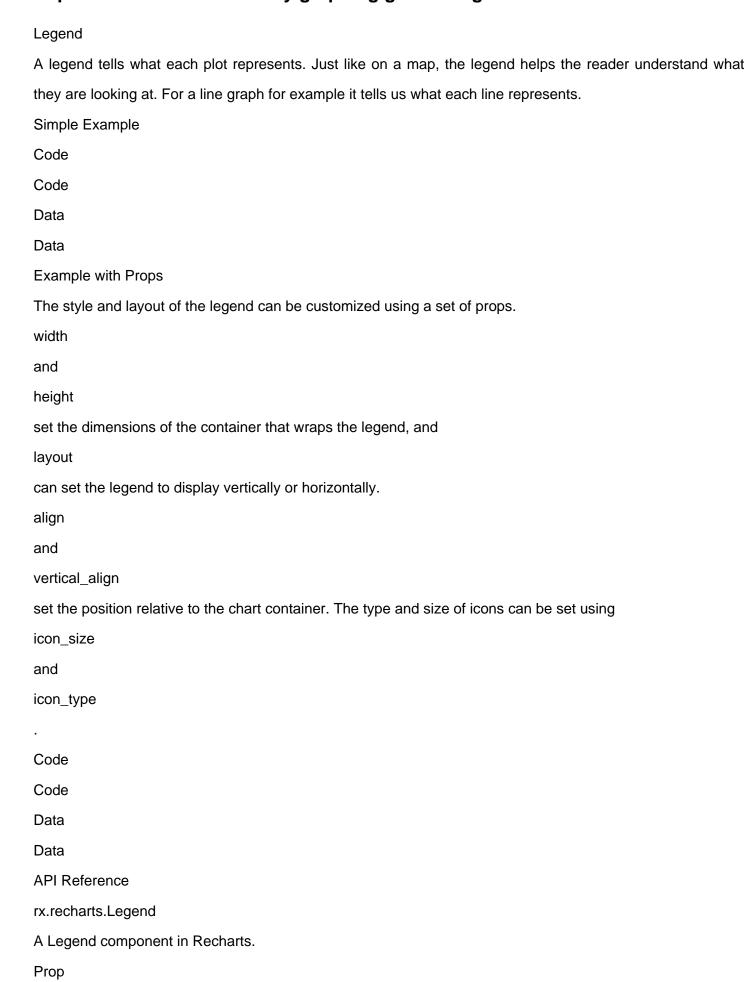
Type | Values

Prop

A Label component in Recharts.

Default
view_box
Dict[str, Any]
value
str
offset
int
position
"top" "left"
Event Triggers
See the full list of default event triggers
rx.recharts.LabelList
A LabelList component in Recharts.
Prop
Type Values
Default
data_key
Union[str, int]
position
"top" "left"
offset
int
5
fill
Union[str, Color]
rx.color("gray", 10)
stroke
Union[str, Color]
"none"
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/graphing/general/legend



```
Type | Values
Default
width
int
height
int
layout
"vertical" | "horizontal"
"horizontal"
align
"left" | "center" | ...
"center"
vertical_align
"top" | "middle" | ...
"bottom"
icon_size
int
14
icon_type
"circle" | "cross" | ...
payload
Sequence
[]
chart_width
int
chart_height
int
margin
Dict[str, Any]
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
```

The customized event handler of click on the	items in this group	

https://reflex.dev/docs/library/graphing/general/reference

Reference

The Reference components in Recharts, including ReferenceLine, ReferenceArea, and ReferenceDot, are used to add visual aids and annotations to the chart, helping to highlight specific data points, ranges, or thresholds for better data interpretation and analysis.

Reference Area

The

rx.recharts.reference_area

component in Recharts is used to highlight a specific area or range on the chart by drawing a rectangular region. It is defined by specifying the coordinates (x1, x2, y1, y2) and can be used to emphasize important data ranges or intervals on the chart.

Code

Code

Data

Data

Reference Line

The

rx.recharts.reference_line

component in rx.recharts is used to draw a horizontal or vertical line on the chart at a specified position. It helps to highlight important values, thresholds, or ranges on the axis, providing visual reference points for better data interpretation.

Code

Code

Data

Data

Reference Dot

The

rx.recharts.reference_dot

component in Recharts is used to mark a specific data point on the chart with a customizable dot. It allows you to highlight important values, outliers, or thresholds by providing a visual reference marker at the specified coordinates (x, y) on the chart.

Code

Code

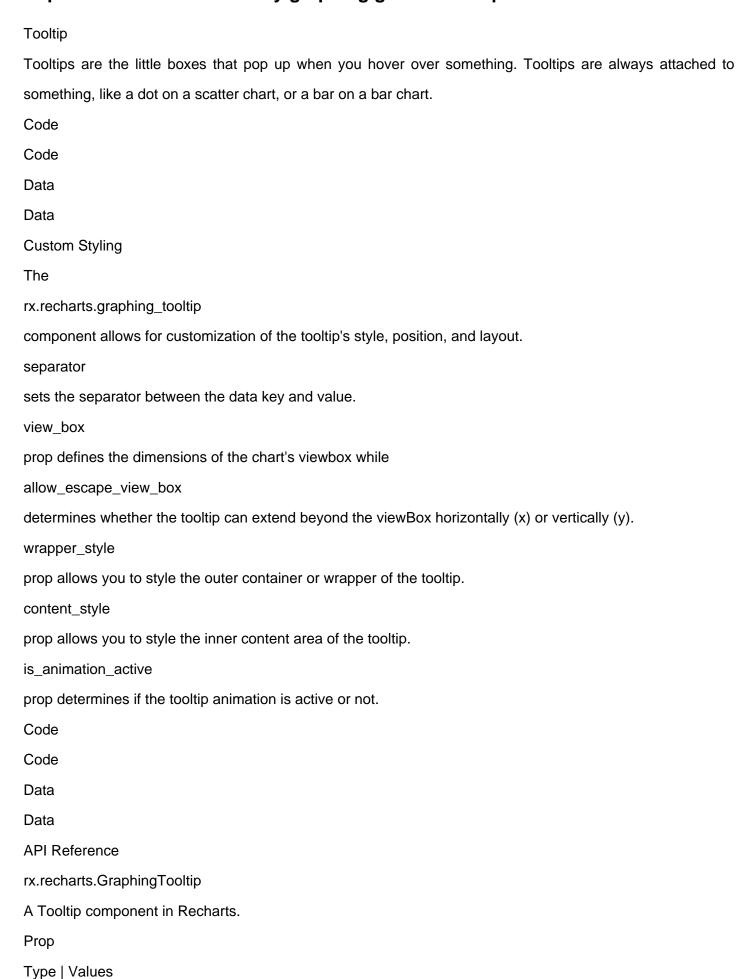
Data
Data
API Reference
rx.recharts.ReferenceLine
A ReferenceLine component in Recharts.
Prop
Type Values
Default
х
Union[str, int]
у
Union[str, int]
stroke
Union[str, Color]
stroke_width
Union[str, int, float]
1
segment
Sequence
x_axis_id
Union[str, int]
0
y_axis_id
Union[str, int]
0
if_overflow
"discard" "hidden"
"discard"
label
Union[str, int]
Valid Children
Label
Event Triggers
See the full list of default event triggers

rx.recharts.ReferenceDot A ReferenceDot component in Recharts. Prop Type | Values Default Χ Union[str, int] Union[str, int] r int fill Union[str, Color] stroke Union[str, Color] x_axis_id Union[str, int] 0 y_axis_id Union[str, int] 0 if_overflow "discard" | "hidden" | ... "discard" label Union[str, int] Valid Children Label **Event Triggers** See the full list of default event triggers Trigger Description on_click

Valid children components The customized event handler of click on the component in this chart

rx.recharts.ReferenceArea A ReferenceArea component in Recharts. Prop Type | Values Default stroke Union[str, Color] fill Union[str, Color] fill_opacity float x_axis_id Union[str, int] y_axis_id Union[str, int] x1 Union[str, int] x2 Union[str, int] y1 Union[str, int] у2 Union[str, int] if_overflow "discard" | "hidden" | ... "discard" Valid Children Label **Event Triggers** See the full list of default event triggers

https://reflex.dev/docs/library/graphing/general/tooltip



```
Default
separator
str
":"
offset
int
10
filter_null
bool
True
cursor
Union[dict, bool]
{"strokeWidth": 1, "fill": rx.color("gray", 3)}
view_box
Dict[str, Any]
item_style
Dict[str, Any]
{"color": rx.color("gray", 12)}
wrapper_style
Dict[str, Any]
{}
content_style
Dict[str, Any]
{"background": rx.color("gray", 1), "borderColor": rx.color("gray", 4), "borderRadius": "8px"}
label_style
Dict[str, Any]
{"color": rx.color("gray", 11)}
allow_escape_view_box
Dict[str, bool]
{"x": False, "y": False}
active
bool
False
position
```

Dict[str, Any]
coordinate
Dict[str, Any]
{"x": 0, "y": 0}
is_animation_active
bool
True
animation_duration
int
1500
animation_easing
"ease" | "ease-in" | ...
"ease"
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/graphing/other-charts

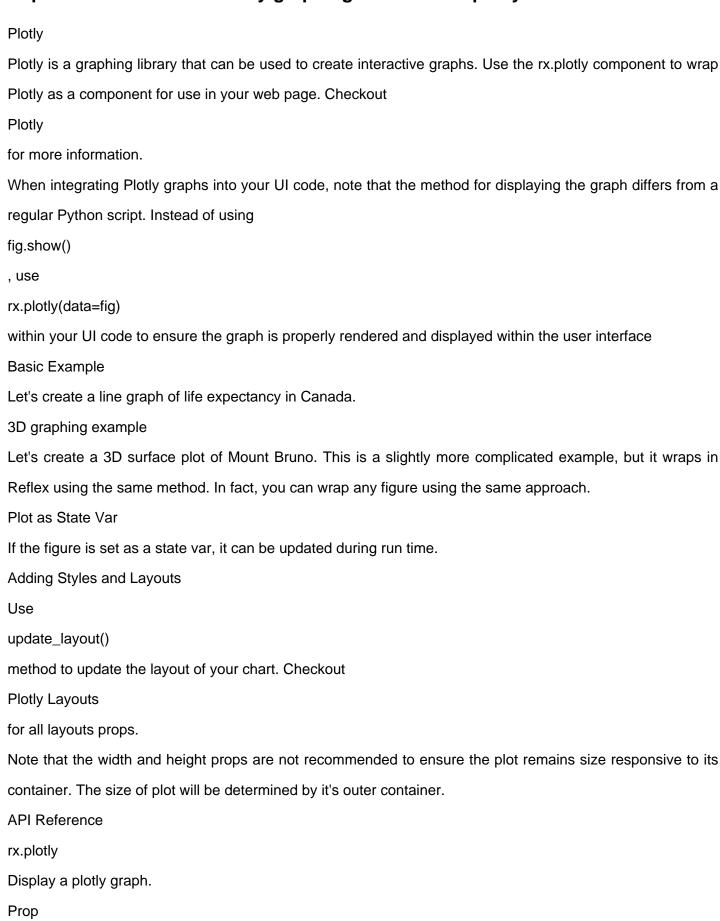
Other Charts

Other graphing components that provide additional functionality and customization options for creating custom visualizations. These components can be used to enhance the graphical representation of data and improve user experience.

Plotly

Pyplot

https://reflex.dev/docs/library/graphing/other-charts/plotly



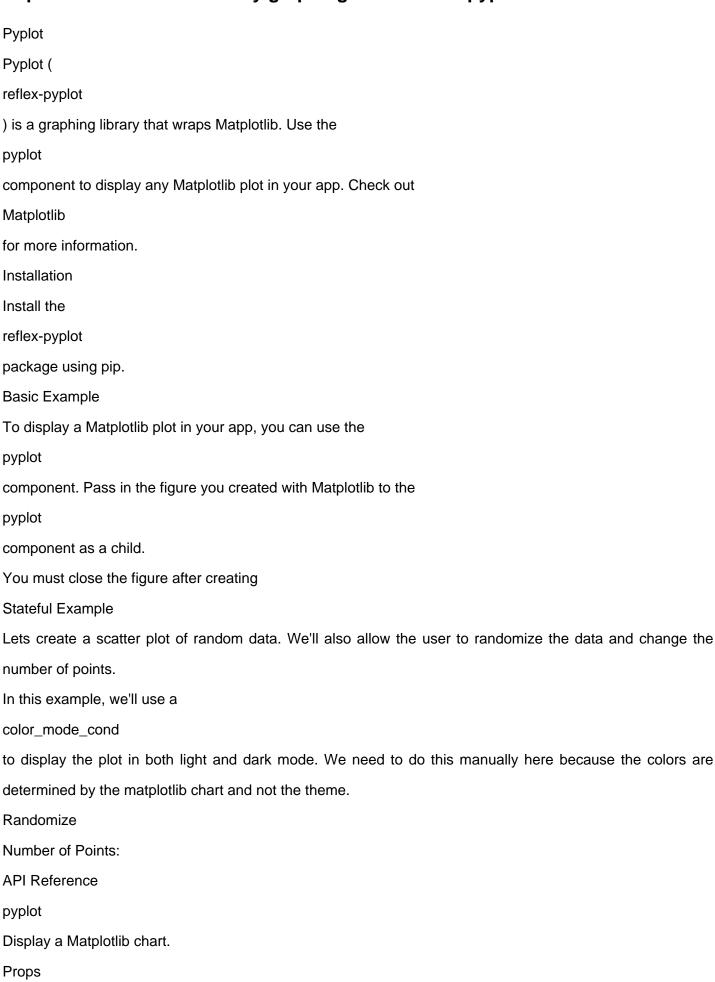
Type | Values

Default

data
Figure
layout
dict
template
Template
config
dict
use_resize_handler
bool
LiteralVar.create(True)
Event Triggers
See the full list of default event triggers
Trigger
Description
on_click
Fired when the plot is clicked.
on_double_click
Fired when the plot is double clicked.
on_after_plot
Fired after the plot is redrawn.
on_animated
Fired after the plot was animated.
on_animating_frame
Fired while animating a single frame (does not currently pass data through).
on_animation_interrupted
Fired when an animation is interrupted (to start a new animation for example).
on_autosize
Fired when the plot is responsively sized.
on_before_hover
Fired whenever mouse moves over a plot.
on_button_clicked
Fired when a plotly UI button is clicked.
on_deselect

Fired when a selection is cleared (via double click). on_hover Fired when a plot element is hovered over. on_relayout Fired after the plot is laid out (zoom, pan, etc). on_relayouting Fired while the plot is being laid out. on_restyle Fired after the plot style is changed. on_redraw Fired after the plot is redrawn. on_selected Fired after selecting plot elements. on_selecting Fired while dragging a selection. on_transitioning Fired while an animation is occurring. on_transition_interrupted Fired when a transition is stopped early. on_unhover Fired when a hovered element is no longer hovered.

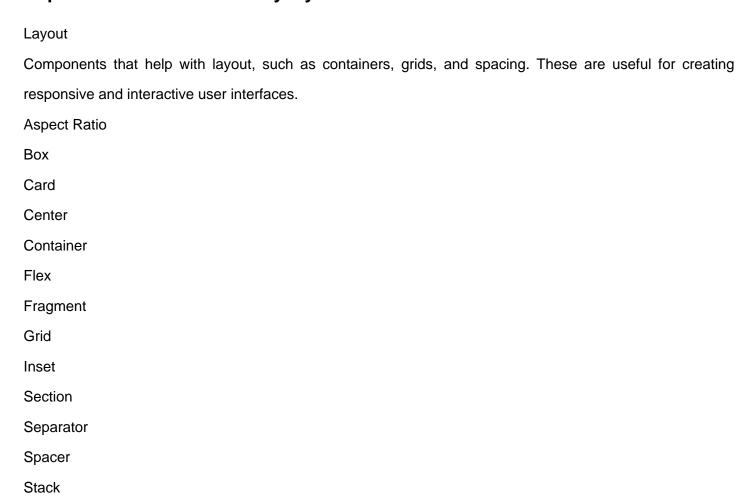
https://reflex.dev/docs/library/graphing/other-charts/pyplot



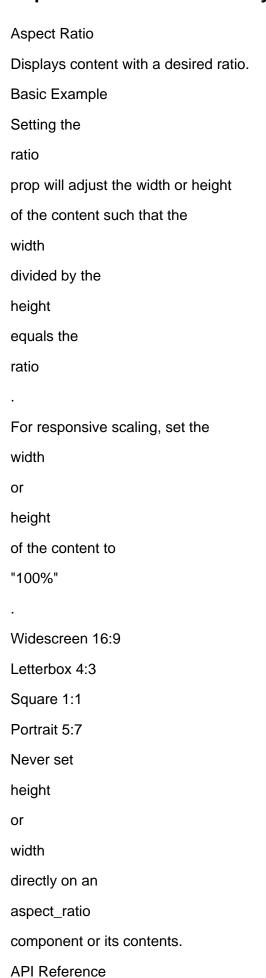
No component specific props

Event Triggers

https://reflex.dev/docs/library/layout

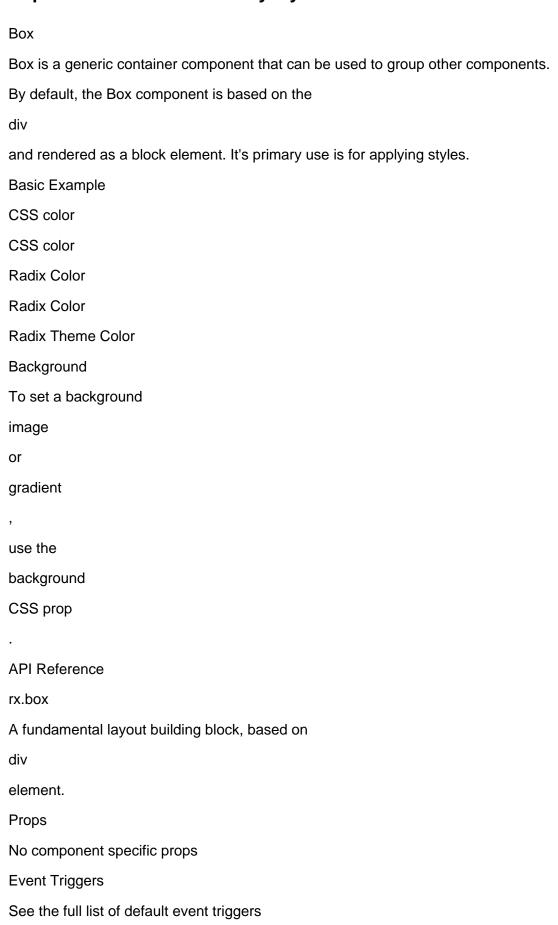


https://reflex.dev/docs/library/layout/aspect-ratio



rx.aspect_ratio
Displays content with a desired ratio.
Test
Prop
Type Values
Default
Interactive
ratio
Union[int, float]
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/layout/box



https://reflex.dev/docs/library/layout/card

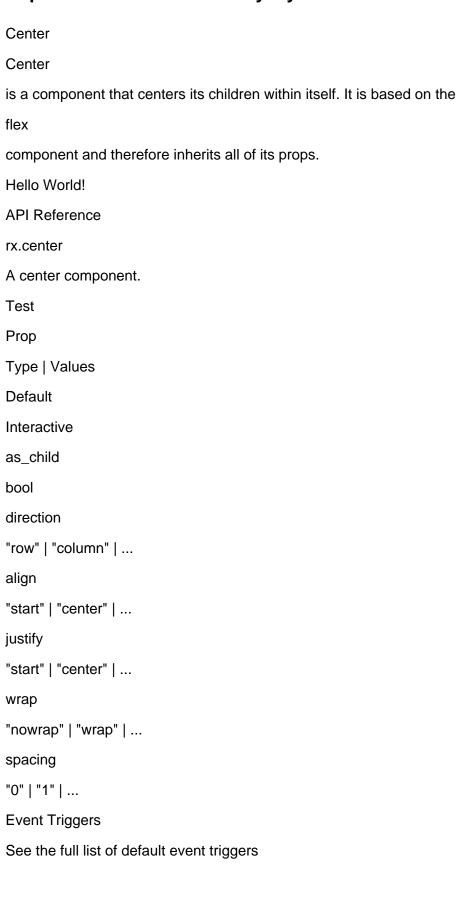
Type | Values

Card A Card component is used for grouping related components. It is similar to the Box, except it has a border, uses the theme colors and border radius, and provides a size prop to control spacing and margin according to the Radix "1" "5" scale. The Card requires less styling than a Box to achieve consistent visual results when used with themes. Basic Example Card 1 Card 2 Card 3 Card 4 Card 5 Rendering as a Different Element The as_child prop may be used to render the Card as a different element. Link and Button are commonly used to make a Card clickable. **Quick Start** Get started with Reflex in 5 minutes. **Using Inset Content API Reference** rx.card Container that groups related content and actions. **Basic Card** Prop

Default
Interactive
as_child
bool
size
"1" | "2" | ...
variant
"surface" | "classic" | ...

Event Triggers

https://reflex.dev/docs/library/layout/center



https://reflex.dev/docs/library/layout/container

Constrains the maximum width of page content, while keeping flexible margins for responsive layouts.

A Container is generally used to wrap the main content for a page.

Basic Example

This content is constrained to a max width of 448px.

This content is constrained to a max width of 688px.

This content is constrained to a max width of 880px.

This content is constrained to a max width of 1136px.

API Reference

rx.container

Constrains the maximum width of page content.

Test

Prop

Type | Values

Default

Interactive

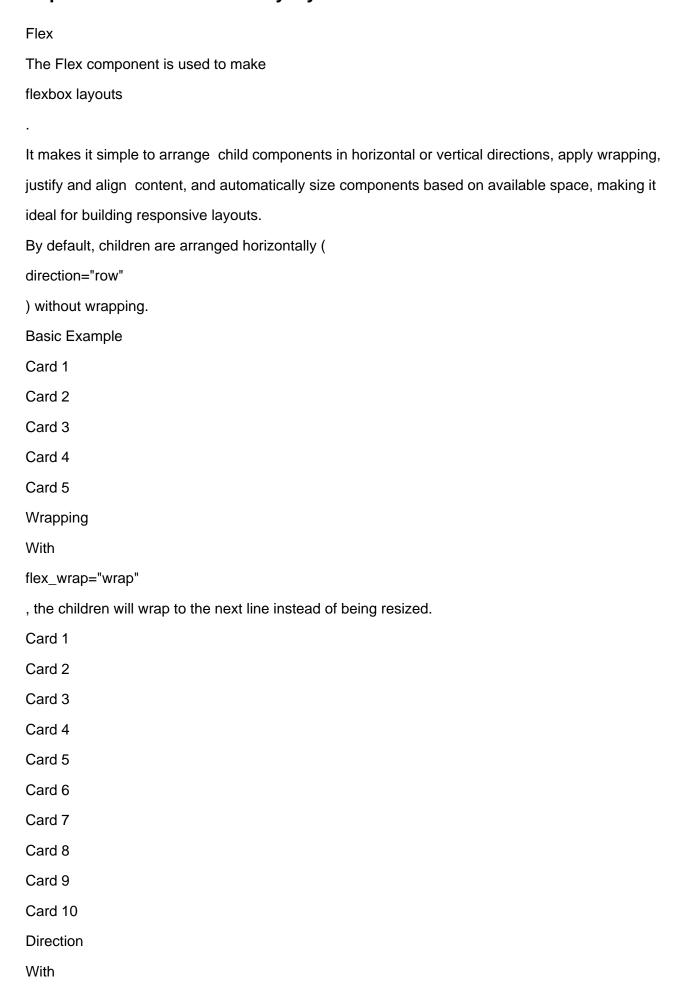
size

"1" | "2" | ...

LiteralVar.create("3")

Event Triggers

https://reflex.dev/docs/library/layout/flex



direction="column"
, the children will be arranged vertically.
Card 1
Card 2
Card 3
Card 4
Alignment
Two props control how children are aligned within the Flex component:
align
controls how children are aligned along the cross axis (vertical for
row
and horizontal for
column
).
justify
controls how children are aligned along the main axis (horizontal for
row
and vertical for
column
).
The following example visually demonstrates the effect of these props with different
wrap
and
direction
values.
Wrap
Direction
Align
Justify
Card 1
Card 2
Card 3
Card 4
Card 5

Card 6
Card 7
Card 8
Card 9
Card 10
Size Hinting
When a child component is included in a flex container,
the
flex_grow
(default
"0"
) and
flex_shrink
(default
"1"
) props control
how the box is sized relative to other components in the same container.
The resizing always applies to the main axis of the flex container. If the direction is
row
10W
, then the sizing applies to the
, then the sizing applies to the
, then the sizing applies to the width
, then the sizing applies to the width . If the direction is
, then the sizing applies to the width . If the direction is column
, then the sizing applies to the width . If the direction is column , then the sizing
, then the sizing applies to the width . If the direction is column , then the sizing applies to the
, then the sizing applies to the width . If the direction is column , then the sizing applies to the height
, then the sizing applies to the width . If the direction is column , then the sizing applies to the height . To set the optimal size along the main axis, the
, then the sizing applies to the width . If the direction is column , then the sizing applies to the height . To set the optimal size along the main axis, the flex_basis
, then the sizing applies to the width . If the direction is column , then the sizing applies to the height . To set the optimal size along the main axis, the flex_basis prop
, then the sizing applies to the width . If the direction is column , then the sizing applies to the height . To set the optimal size along the main axis, the flex_basis prop is used and may be either a percentage or CSS size units. When unspecified, the
, then the sizing applies to the width . If the direction is column , then the sizing applies to the height . To set the optimal size along the main axis, the flex_basis prop is used and may be either a percentage or CSS size units. When unspecified, the corresponding

```
value is used if set, otherwise the content size is used.
When
flex_grow="0"
, the box will not grow beyond the
flex_basis
When
flex_shrink="0"
, the box will not shrink to less than the
flex_basis
These props are used when creating flexible responsive layouts.
Move the slider below and see how adjusting the width of the flex container
affects the computed sizes of the flex items based on the props that are set.
flex shrink=0
flex_shrink=1
flex_grow=0
flex_grow=1
API Reference
rx.flex
Component for creating flex layouts.
Test
Prop
Type | Values
Default
Interactive
as_child
bool
direction
"row" | "column" | ...
align
"start" | "center" | ...
justify
"start" | "center" | ...
```

```
wrap
"nowrap" | "wrap" | ...
spacing
"0" | "1" | ...
```

Event Triggers

https://reflex.dev/docs/library/layout/fragment



A Fragment is a Component that allow you to group multiple Components without a wrapper node.

Refer to the React docs at

React/Fragment

for more information on its use-case.

Component1

Component2

Video: Fragment

API Reference

rx.fragment

A React fragment to return multiple components from a function without wrapping it in a container.

Props

No component specific props

Event Triggers

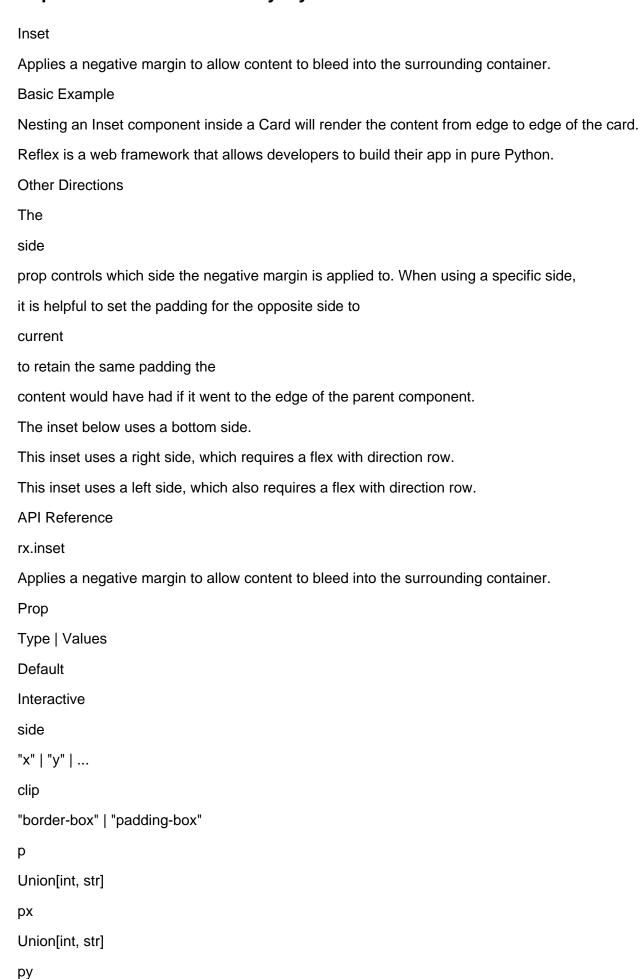
https://reflex.dev/docs/library/layout/grid

Grid	
Component for creating grid layouts. Either	
rows	
or	
columns	
may be specified.	
Basic Example	
Card 1	
Card 2	
Card 3	
Card 4	
Card 5	
Card 6	
Card 7	
Card 8	
Card 9	
Card 10	
Card 11	
Card 12	
Card 1	
Card 2	
Card 3	
Card 4	
Card 5	
Card 6	
Card 7	
Card 8	
Card 9	
Card 10	
Card 11	
Card 12	

API Reference

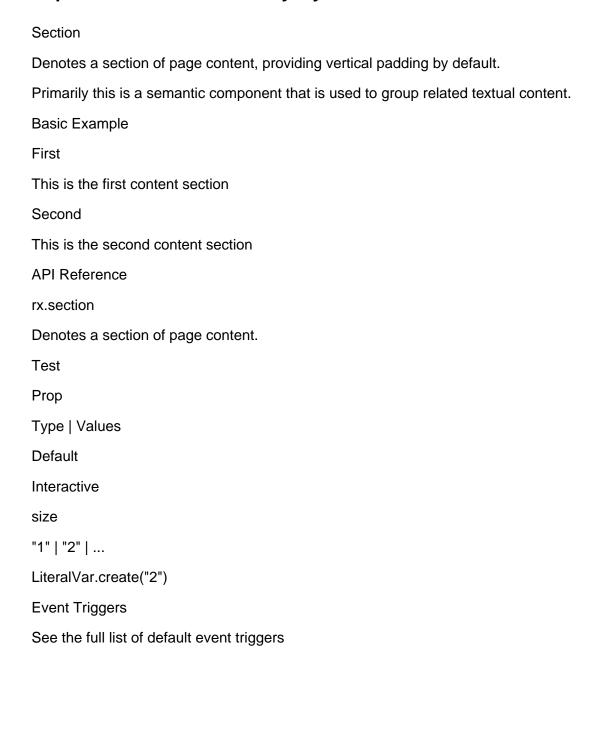
rx.grid
Component for creating grid layouts.
Test
Prop
Type Values
Default
Interactive
as_child
bool
columns
str
rows
str
flow
"row" "column"
align
"start" "center"
justify
"start" "center"
spacing
"0" "1"
spacing_x
"0" "1"
spacing_y
"0" "1"
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/layout/inset

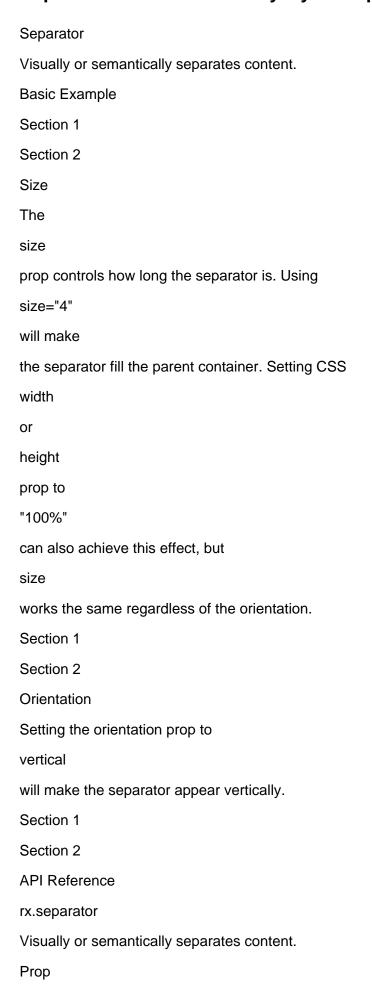


Union[int, str]
pt
Union[int, str]
pr
Union[int, str]
pb
Union[int, str]
pl
Union[int, str]
Event Triggers

https://reflex.dev/docs/library/layout/section



https://reflex.dev/docs/library/layout/separator



Type Values
Default
Interactive
size
"1" "2"
LiteralVar.create("4")
color_scheme
"tomato" "red"
tomato
orientation
"horizontal" "vertical"
decorative
bool
false
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/layout/spacer

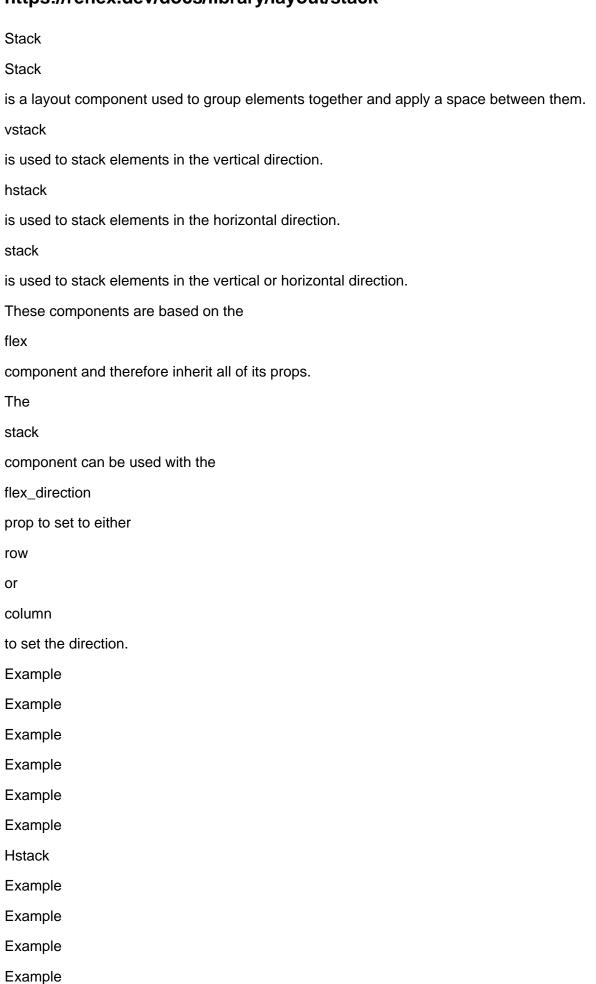


```
justify
"start" | "center" | ...
wrap
"nowrap" | "wrap" | ...
spacing
"0" | "1" | ...
```

Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/layout/stack



Example
Vstack
Example
Real World Example
Saving Money
Saving money is an art that combines discipline, strategic planning, and the wisdom to foresee future needs
and emergencies. It begins with the simple act of setting aside a portion of one's income, creating a buffer
that can grow over time through interest or investments.
Spending Money
Spending money is a balancing act between fulfilling immediate desires and maintaining long-term financial
health. It's about making choices, sometimes indulging in the pleasures of the moment, and at other times,
prioritizing essential expenses.
API Reference
rx.stack
A stack component.
Card 1
Card 2
Card 3
Prop
Type Values
Default
Interactive
spacing
"0" "1"
Var.create("3")
align
"start" "center"
Var.create("start")
as_child
bool

```
direction
"row" | "column" | ...
justify
"start" | "center" | ...
wrap
"nowrap" | "wrap" | ...
Event Triggers
See the full list of default event triggers
rx.hstack
A horizontal stack component.
Test
Prop
Type | Values
Default
Interactive
direction
"row" | "column" | ...
Var.create("row")
spacing
"0" | "1" | ...
Var.create("3")
align
"start" | "center" | ...
Var.create("start")
as_child
bool
justify
"start" | "center" | ...
wrap
"nowrap" | "wrap" | ...
Event Triggers
See the full list of default event triggers
rx.vstack
```

A vertical stack component.

```
Test
Prop
Type | Values
Default
Interactive
direction
"row" | "column" | ...
Var.create("column")
spacing
"0" | "1" | ...
Var.create("3")
align
"start" | "center" | ...
Var.create("start")
as_child
bool
justify
"start" | "center" | ...
wrap
"nowrap" | "wrap" | ...
Event Triggers
See the full list of default event triggers
```

https://reflex.dev/docs/library/media

Media

Components that help with media, such as images, videos, and audio. These are useful for creating responsive and interactive user interfaces.

Audio

Image

Video

https://reflex.dev/docs/library/media/audio

muted

bool

Audio The audio component can display an audio given an src path as an argument. This could either be a local path from the assets folder or an external link. If we had a local file in the assets folder named test.mp3 we could set url="/test.mp3" to view the audio file. How to let your user upload an audio file **API Reference** rx.audio Audio component share with Video component. Prop Type | Values Default url str playing bool loop bool controls bool Var.create(True) light bool volume float

Event Triggers See the full list of default event triggers Trigger Description on_ready Called when media is loaded and ready to play. If playing is set to true, media will play immediately. on_start Called when media starts playing. on_play Called when media starts or resumes playing after pausing or buffering. on_progress Callback containing played and loaded progress as a fraction, and playedSeconds and loadedSeconds in seconds. eg { played: 0.12, playedSeconds: 11.3, loaded: 0.34, loadedSeconds: 16.7 } on_duration Callback containing duration of the media, in seconds. on_pause Called when media is paused. on_buffer Called when media starts buffering. on_buffer_end Called when media has finished buffering. Works for files, YouTube and Facebook. on_seek Called when media seeks with seconds parameter. on_playback_rate_change Called when playback rate of the player changed. Only supported by YouTube, Vimeo (if enabled), Wistia, and file paths. on playback quality change Called when playback quality of the player changed. Only supported by YouTube (if enabled). on_ended Called when media finishes playing. Does not fire when loop is set to true. on_error

Called when an error occurs whilst attempting to play media.

Called when user clicks the light mode preview.

on_click_preview

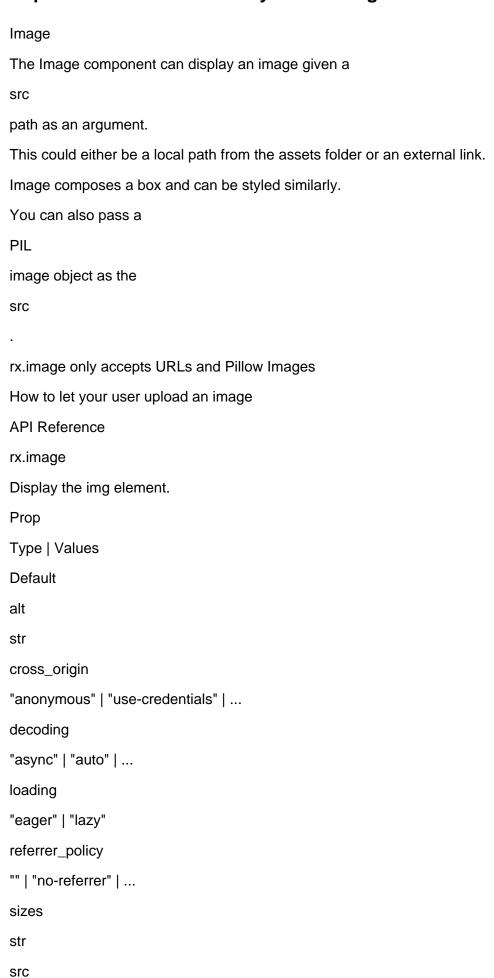
on_enable_pip

Called when picture-in-picture mode is enabled.

on_disable_pip

Called when picture-in-picture mode is disabled.

https://reflex.dev/docs/library/media/image



Any
src_set
str
use_map
str

Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/media/video

muted

bool

Video The video component can display a video given an src path as an argument. This could either be a local path from the assets folder or an external link. If we had a local file in the assets folder named test.mp4 we could set url="/test.mp4" to view the video. How to let your user upload a video **API Reference** rx.video Video component share with audio component. Prop Type | Values Default url str playing bool loop bool controls bool Var.create(True) light bool volume float

Event Triggers See the full list of default event triggers Trigger Description on_ready Called when media is loaded and ready to play. If playing is set to true, media will play immediately. on_start Called when media starts playing. on_play Called when media starts or resumes playing after pausing or buffering. on_progress Callback containing played and loaded progress as a fraction, and playedSeconds and loadedSeconds in seconds. eg { played: 0.12, playedSeconds: 11.3, loaded: 0.34, loadedSeconds: 16.7 } on_duration Callback containing duration of the media, in seconds. on_pause Called when media is paused. on_buffer Called when media starts buffering. on_buffer_end Called when media has finished buffering. Works for files, YouTube and Facebook. on_seek Called when media seeks with seconds parameter. on_playback_rate_change Called when playback rate of the player changed. Only supported by YouTube, Vimeo (if enabled), Wistia, and file paths. on playback quality change Called when playback quality of the player changed. Only supported by YouTube (if enabled). on_ended Called when media finishes playing. Does not fire when loop is set to true. on_error

Called when an error occurs whilst attempting to play media.

Called when user clicks the light mode preview.

on_click_preview

on_enable_pip

Called when picture-in-picture mode is enabled.

on_disable_pip

Called when picture-in-picture mode is disabled.

https://reflex.dev/docs/library/overlay

Overlay													
Components	that help	with	overlays,	such	as	modals,	popovers,	and tooltips.	These	are use	eful fo	r c	reating

responsive and interactive user interfaces.

Context Menu

Alert Dialog

Dialog

Drawer

Dropdown Menu

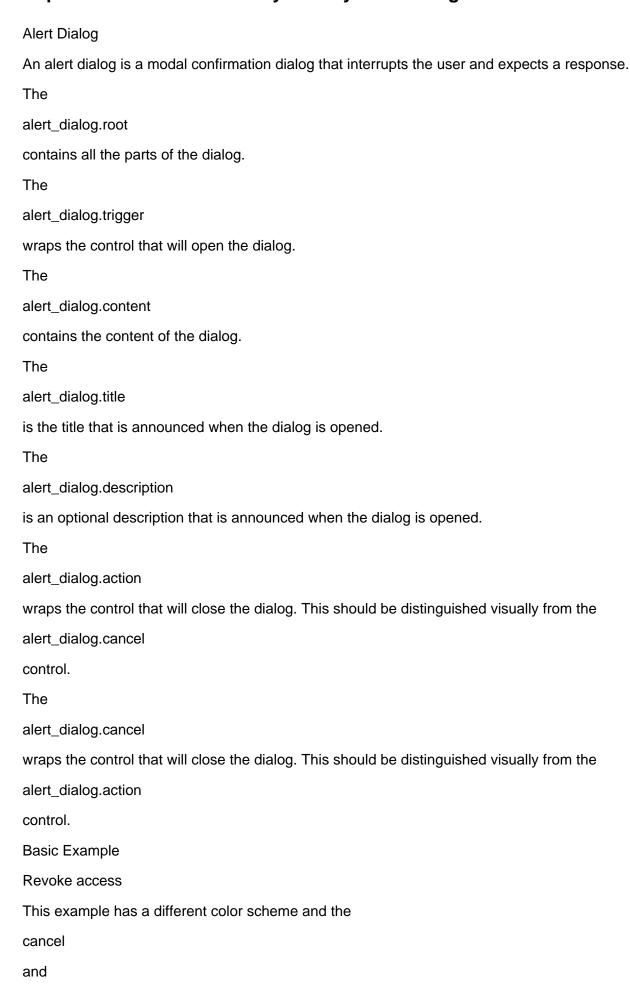
Hover Card

Popover

Toast

Tooltip

https://reflex.dev/docs/library/overlay/alert-dialog



action
buttons are right aligned.
Revoke access
Use the
inset
component to align content flush with the sides of the dialog.
Delete Users
Events when the Alert Dialog opens or closes
The
on_open_change
event is called when the
open
state of the dialog changes. It is used in conjunction with the
open
prop.
Number of times alert dialog opened or closed: 0
Alert Dialog open: false
Revoke access
Controlling Alert Dialog with State
This example shows how to control whether the dialog is open or not with state. This is an easy way to show
the dialog without needing to use the
rx.alert_dialog.trigger
•
rx.alert_dialog.root
has a prop
open
that can be set to a boolean value to control whether the dialog is open or not.
We toggle this
open
prop with a button outside of the dialog and the
rx.alert_dialog.cancel
and
rx.alert_dialog.action
buttons inside the dialog.

Button to Open the Dialog Form Submission to a Database from an Alert Dialog This example adds new users to a database from an alert dialog using a form. It defines a User1 model with name and email fields. The add_user_to_db method adds a new user to the database, checking for existing emails. On form submission, it calls the add_user_to_db method. The UI component has: A button to open an alert dialog An alert dialog containing a form to add a new user Input fields for name and email Submit and Cancel buttons Add User **API Reference** rx.alert_dialog.root Contains all the parts of the dialog. Revoke access Prop Type | Values Default Interactive open bool default_open bool **Event Triggers** See the full list of default event triggers Trigger Description on_open_change Fired when the open state changes.

rx.alert_dialog.content Contains the content of the dialog. This component is based on the div element. Revoke access Prop Type | Values Default Interactive size "1" | "2" | ... force_mount bool false **Event Triggers** See the full list of default event triggers Trigger Description on_open_auto_focus Fired when the dialog is opened. on_close_auto_focus Fired when the dialog is closed. on_escape_key_down Fired when the escape key is pressed. rx.alert_dialog.trigger Wraps the control that will open the dialog. Props No component specific props **Event Triggers** See the full list of default event triggers rx.alert_dialog.title An accessible title that is announced when the dialog is opened. This part is based on the Heading component with a pre-defined font size and leading trim on top. Props No component specific props

Event Triggers

See the full list of default event triggers

rx.alert_dialog.description

An optional accessible description that is announced when the dialog is opened.

This part is based on the Text component with a pre-defined font size.

Props

No component specific props

Event Triggers

See the full list of default event triggers

rx.alert_dialog.action

Wraps the control that will close the dialog. This should be distinguished

visually from the Cancel control.

Props

No component specific props

Event Triggers

See the full list of default event triggers

rx.alert_dialog.cancel

Wraps the control that will close the dialog. This should be distinguished

visually from the Action control.

Props

No component specific props

Event Triggers

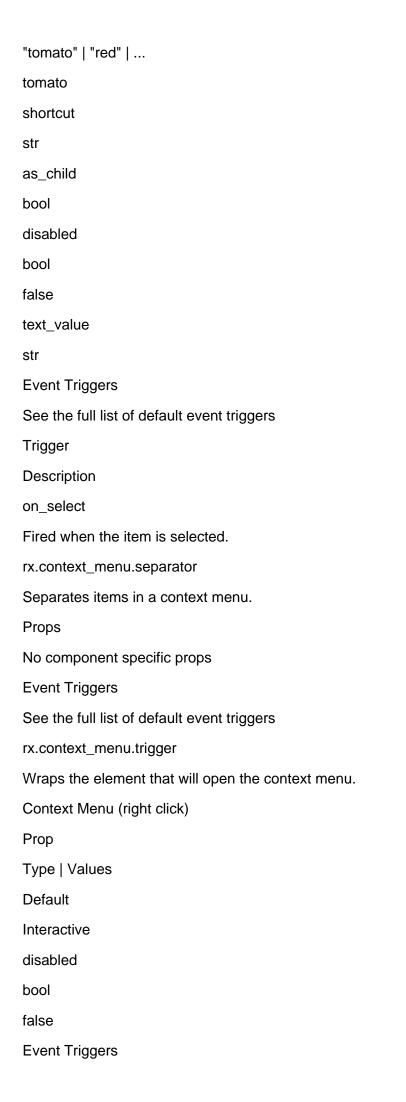
See the full list of default event triggers

https://reflex.dev/docs/library/overlay/context-menu

context_menu.sub

Context Menu A Context Menu is a popup menu that appears upon user interaction, such as a right-click or a hover. **Basic Usage** A Context Menu is composed of a context_menu.root , a context_menu.trigger and a context_menu.content . The context_menu_root contains all the parts of a context menu. The context_menu.trigger is the element that the user interacts with to open the menu. It wraps the element that will open the context menu. The context_menu.content is the component that pops out when the context menu is open. The context menu.item contains the actual context menu items and sits under the context_menu.content The context_menu.sub contains all the parts of a submenu. There is a context_menu.sub_trigger , which is an item that opens a submenu. It must be rendered inside a context_menu.sub component. The context_menu.sub_content is the component that pops out when a submenu is open. It must also be rendered inside a

component.
The
context_menu.separator
is used to visually separate items in a context menu.
Right click me
In this example, we will show how to open a dialog box from a context menu, where the menu will close and
the dialog will open and be functional.
API Reference
rx.context_menu.root
Menu representing a set of actions, displayed at the origin of a pointer right-click or long-press.
Context Menu (right click)
Prop
Type Values
Default
Interactive
modal
bool
false
dir
"ltr" "rtl"
Event Triggers
See the full list of default event triggers
Trigger
Description
on_open_change
Fired when the open state changes.
rx.context_menu.item
The component that contains the context menu items.
Context Menu (right click)
Prop
Type Values
Default
Interactive
color_scheme



See the full list of default event triggers rx.context_menu.content The component that pops out when the context menu is open. Context Menu (right click) Prop Type | Values Default Interactive size "1" | "2" variant "solid" | "soft" color_scheme "tomato" | "red" | ... tomato high_contrast bool false as_child bool loop bool false force_mount bool false side "top" | "right" | ... side_offset Union[int, float] align "start" | "center" | ... align_offset

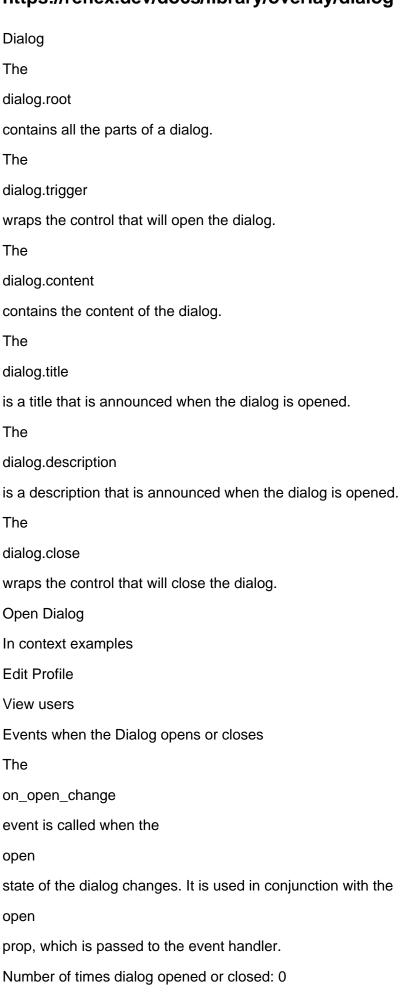
Union[int, float]

avoid_collisions
bool
false
collision_padding
Union[float, int, dict]
sticky
"partial" "always"
hide_when_detached
bool
false
Event Triggers
See the full list of default event triggers
Trigger
Description
on_close_auto_focus
Fired when focus moves back after closing.
on_escape_key_down
Fired when the escape key is pressed.
on_pointer_down_outside
Fired when a pointer down event happens outside the context menu.
on_focus_outside
Fired when focus moves outside the context menu.
on_interact_outside
Fired when the pointer interacts outside the context menu.
rx.context_menu.sub
Contains all the parts of a submenu.
Context Menu (right click)
Prop
Type Values
Default
Interactive
open
bool
default_open

bool
Event Triggers
See the full list of default event triggers
Trigger
Description
on_open_change
Fired when the open state changes.
rx.context_menu.sub_trigger
An item that opens a submenu.
Context Menu (right click)
Prop
Type Values
Default
Interactive
as_child
bool
disabled
bool
false
text_value
str
Event Triggers
See the full list of default event triggers
rx.context_menu.sub_content
The component that pops out when a submenu is open.
Context Menu (right click)
Prop
Type Values
Default
Interactive
as_child
bool
Іоор
bool

false
force_mount
bool
false
side_offset
Union[int, float]
align_offset
Union[int, float]
avoid_collisions
bool
false
collision_padding
Union[float, int, dict]
sticky
"partial" "always"
hide_when_detached
bool
false
Event Triggers
See the full list of default event triggers
Trigger
Description
on_escape_key_down
Fired when the escape key is pressed.
on_pointer_down_outside
Fired when a pointer down event happens outside the context menu.
on_focus_outside
Fired when focus moves outside the context menu.
on_interact_outside
Fired when interacting outside the context menu.

https://reflex.dev/docs/library/overlay/dialog



Dialog open: false
Open Dialog
Check out the
menu docs
for an example of opening a dialog from within a dropdown menu.
Form Submission to a Database from a Dialog
This example adds new users to a database from a dialog using a form.
It defines a User model with name and email fields.
The
add_user_to_db
method adds a new user to the database, checking for existing emails.
On form submission, it calls the
add_user_to_db
method.
The UI component has:
A button to open a dialog
A dialog containing a form to add a new user
Input fields for name and email
Submit and Cancel buttons
Add User
API Reference
rx.dialog.root
Root component for Dialog.
Open Dialog
Prop
Type Values
Default
Interactive
open
bool
default_open
bool
Event Triggers
See the full list of default event triggers

Trigger
Description
on_open_change
Fired when the open state changes.
rx.dialog.trigger
Trigger an action or event, to open a Dialog modal.
Props
No component specific props
Event Triggers
See the full list of default event triggers
rx.dialog.title
Title component to display inside a Dialog modal.
Props
No component specific props
Event Triggers
See the full list of default event triggers
rx.dialog.content
Content component to display inside a Dialog modal.
Open Dialog
Prop
Type Values
Default
Interactive
size
"1" "2"
Event Triggers
See the full list of default event triggers
Trigger
Description
on_open_auto_focus
Fired when the dialog is opened.
on_close_auto_focus
Fired when the dialog is closed.
on_escape_key_down

Fired when the escape key is pressed.

on_pointer_down_outside

Fired when the pointer is down outside the dialog.

on_interact_outside

Fired when the pointer interacts outside the dialog.

rx.dialog.description

Description component to display inside a Dialog modal.

Props

No component specific props

Event Triggers

See the full list of default event triggers

rx.dialog.close

Close button component to close an open Dialog modal.

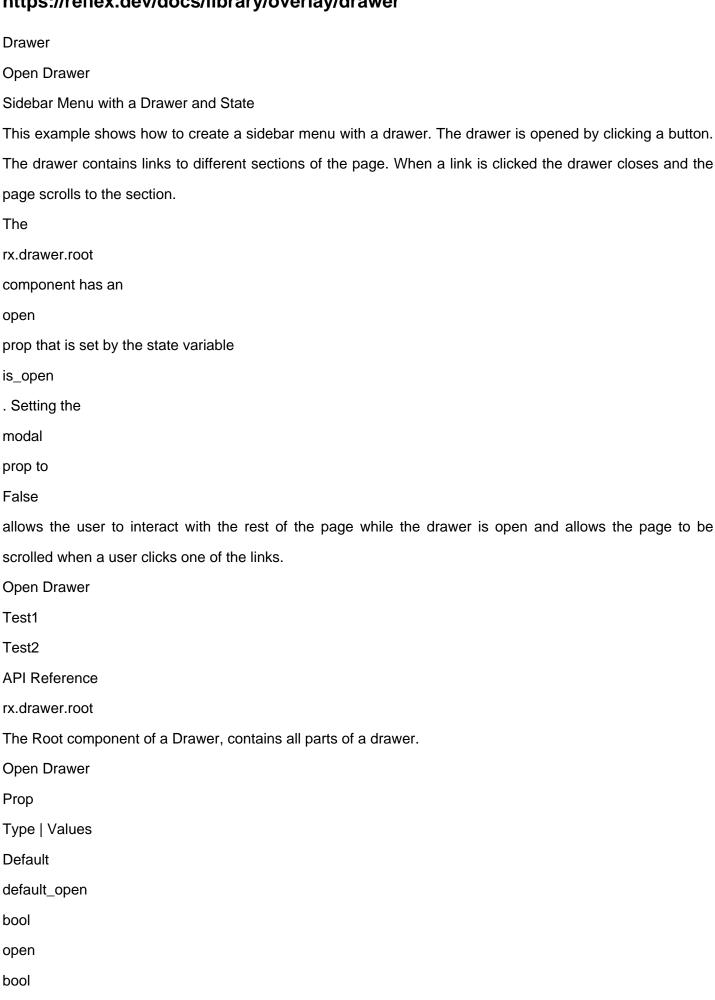
Props

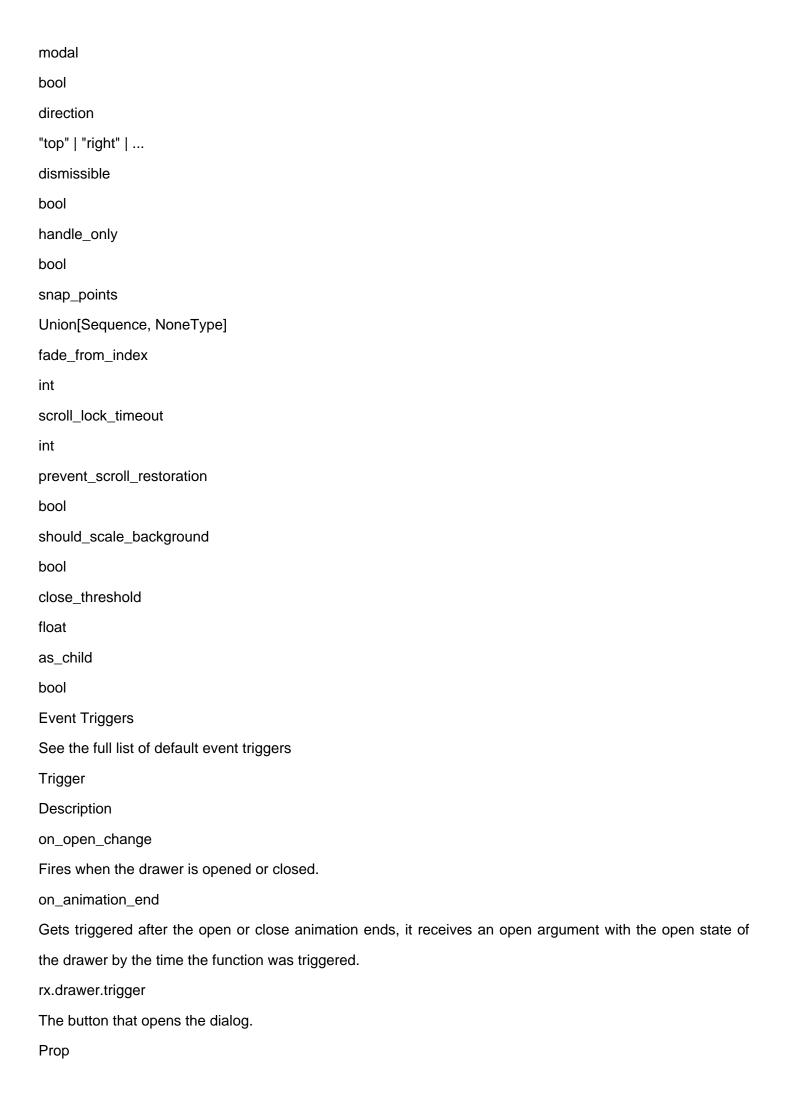
No component specific props

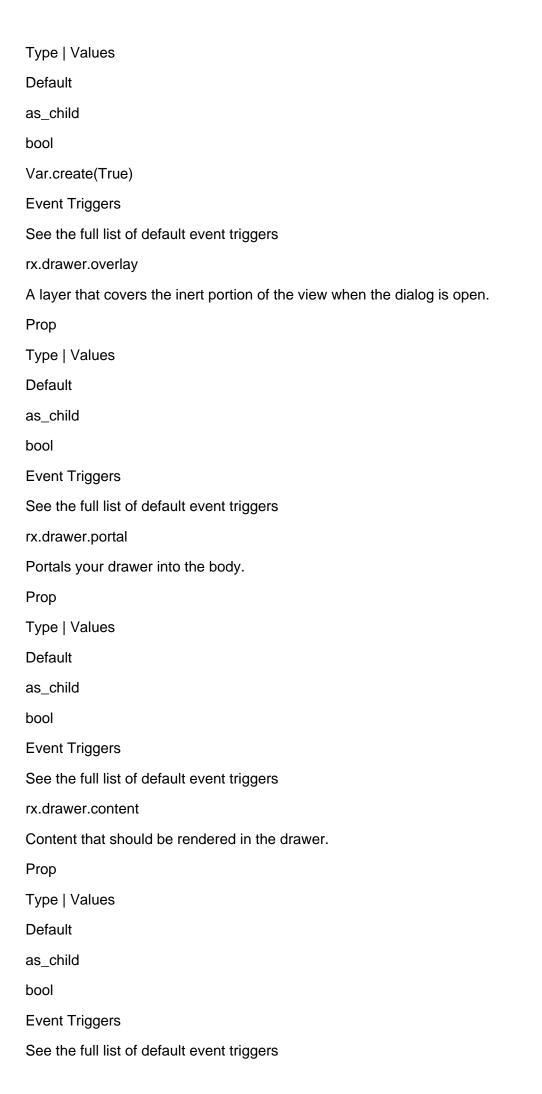
Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/overlay/drawer







Trigger

Description

on_open_auto_focus

The on_open_auto_focus event handler is called when the component opens and the focus is returned to the first item.

on_close_auto_focus

The on_close_auto_focus event handler is called when focus moves to the trigger after closing. It can be prevented by calling event.preventDefault.

on_escape_key_down

The on_escape_key_down event handler is called when the escape key is down. It can be prevented by calling event.preventDefault.

on_pointer_down_outside

The on_pointer_down_outside event handler is called when a pointer event occurs outside the bounds of the component. It can be prevented by calling event.preventDefault.

on_interact_outside

The on_interact_outside event handler is called when the user interacts outside the component.

rx.drawer.close

A button that closes the drawer.

Prop

Type | Values

Default

as_child

bool

Var.create(True)

Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/overlay/dropdown-menu

Dropdown Menu

A Dropdown Menu is a menu that offers a list of options that a user can select from. They are typically positioned near a button that will control their appearance and disappearance.

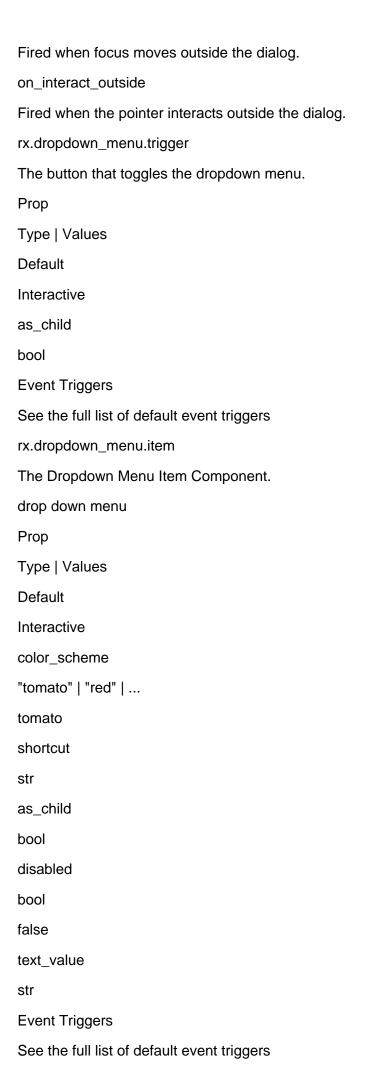
A Dropdown Menu is composed of a menu.root , a menu.trigger and a menu.content . The menu.trigger is the element that the user interacts with to open the menu. It wraps the element that will open the dropdown menu. The menu.content is the component that pops out when the dropdown menu is open. The menu.item contains the actual dropdown menu items and sits under the menu.content . The shortcut prop is an optional shortcut command displayed next to the item text. The menu.sub contains all the parts of a submenu. There is a menu.sub trigger , which is an item that opens a submenu. It must be rendered inside a menu.sub component. The menu.sub_component

is the component that pops out when a submenu is open. It must also be rendered inside a menu.sub

component.
The
menu.separator
is used to visually separate items in a dropdown menu.
Options
Events when the Dropdown Menu opens or closes
The
on_open_change
event, from the
menu.root
, is called when the
open
state of the dropdown menu changes. It is used in conjunction with the
open
prop, which is passed to the event handler.
Number of times Dropdown Menu opened or closed: 0
Dropdown Menu open: false
Options
Opening a Dialog from Menu using State
Accessing an overlay component from within another overlay component is a common use case but does not
always work exactly as expected.
The code below will not work as expected as because the dialog is within the menu and the dialog will only
be open when the menu is open, rendering the dialog unusable.
In this example, we will show how to open a dialog box from a dropdown menu, where the menu will close
and the dialog will open and be functional.
API Reference
rx.dropdown_menu.root
The Dropdown Menu Root Component.
drop down menu
Prop
Type Values
Default
Interactive
default_open

bool
open
bool
modal
bool
false
dir
"ltr" "rtl"
Event Triggers
See the full list of default event triggers
Trigger
Description
on_open_change
Fired when the open state changes.
rx.dropdown_menu.content
The Dropdown Menu Content component that pops out when the dropdown menu is open.
drop down menu
Prop
Type Values
Default
Interactive
size
"1" "2"
variant
"solid" "soft"
color_scheme
"tomato" "red"
tomato
high_contrast
bool
false
as_child
as_criliu
bool

```
bool
false
force_mount
bool
false
side
"top" | "right" | ...
side_offset
Union[int, float]
align
"start" | "center" | ...
align_offset
Union[int, float]
avoid_collisions
bool
false
collision_padding
Union[float, int, dict]
sticky
"partial" | "always"
hide_when_detached
bool
false
Event Triggers
See the full list of default event triggers
Trigger
Description
on_close_auto_focus
Fired when the dialog is closed.
on_escape_key_down
Fired when the escape key is pressed.
on_pointer_down_outside
Fired when the pointer is down outside the dialog.
on_focus_outside
```



Trigger
Description
on_select
Fired when the item is selected.
rx.dropdown_menu.separator
Dropdown Menu Separator Component. Used to visually separate items in the dropdown menu.
Props
No component specific props
Event Triggers
See the full list of default event triggers
rx.dropdown_menu.sub_content
The component that pops out when a submenu is open. Must be rendered inside DropdownMenuSub.
drop down menu
Prop
Type Values
Default
Interactive
as_child
bool
Іоор
bool
false
force_mount
bool
false
side_offset
Union[int, float]
align_offset
Union[int, float]
avoid_collisions
bool
false
collision_padding
Union[float, int, dict]

```
sticky
"partial" | "always"
hide_when_detached
bool
false
Event Triggers
See the full list of default event triggers
Trigger
Description
on_escape_key_down
Fired when the escape key is pressed.
on_pointer_down_outside
Fired when the pointer is down outside the dialog.
on_focus_outside
Fired when focus moves outside the dialog.
on_interact_outside
Fired when the pointer interacts outside the dialog.
```

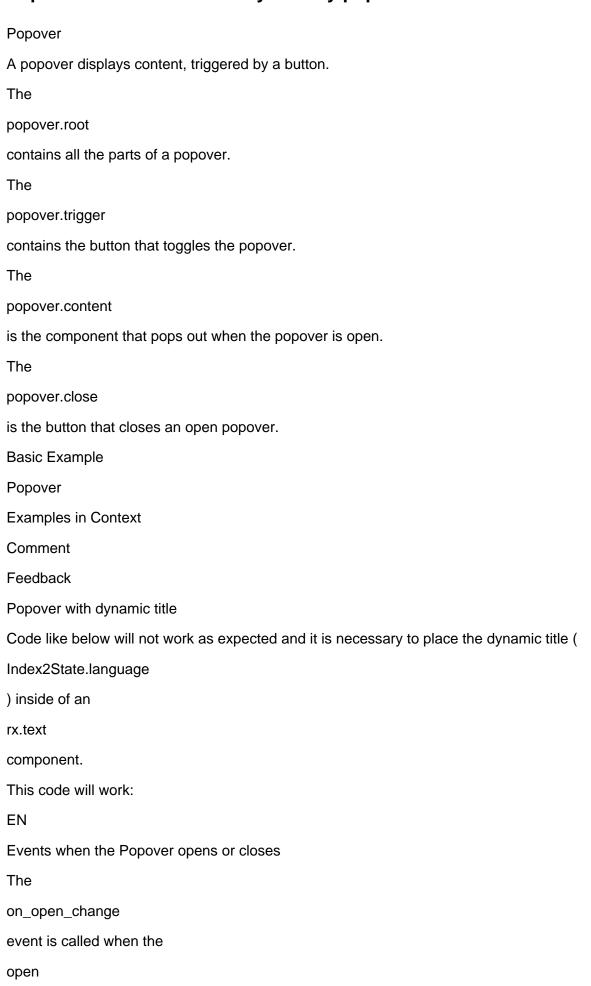
https://reflex.dev/docs/library/overlay/hover-card

https://reflex.dev/docs/library/overlay/hover-card
Hovercard
The
hover_card.root
contains all the parts of a hover card.
The
hover_card.trigger
wraps the link that will open the hover card.
The
hover_card.content
contains the content of the open hover card.
Hover over the text to see the tooltip.
Hover over me
Hover over the text to see the tooltip.
Hover over me
Events when the Hovercard opens or closes
The
on_open_change
event is called when the
open
state of the hovercard changes. It is used in conjunction with the
open
prop, which is passed to the event handler.
Number of times hovercard opened or closed: 0
Hovercard open: false
Hover over the text to see the hover card.
Hover over me
API Reference
rx.hover_card.root
For sighted users to preview content available behind a link.
Hover over me
Prop
Type Values

Default
Interactive
default_open
bool
open
bool
open_delay
int
close_delay
int
Event Triggers
See the full list of default event triggers
Trigger
Description
on_open_change
Fired when the open state changes.
rx.hover_card.content
Contains the content of the open hover card.
Hover over me
Prop
Type Values
Default
Interactive
side
"top" "right"
side_offset
int
align
"start" "center"
align_offset
int
avoid_collisions
bool
false

collision_padding Union[float, int, dict] sticky "partial" | "always" hide_when_detached bool false size "1" | "2" | ... **Event Triggers** See the full list of default event triggers rx.hover_card.trigger Wraps the link that will open the hover card. **Props** No component specific props **Event Triggers** See the full list of default event triggers

https://reflex.dev/docs/library/overlay/popover



state of the popover changes. It is used in conjunction with the
open
prop, which is passed to the event handler.
Number of times popover opened or closed: 0
Popover open: false
Popover
API Reference
rx.popover.root
Floating element for displaying rich content, triggered by a button.
Popover
Prop
Type Values
Default
Interactive
open
bool
modal
bool
false
default_open
bool
Event Triggers
See the full list of default event triggers
Trigger
Description
on_open_change
Fired when the open state changes.
rx.popover.content
Contains content to be rendered in the open popover.
Popover
Prop
Type Values
Default
Interactive

```
size
"1" | "2" | ...
side
"top" | "right" | ...
side_offset
int
align
"start" | "center" | ...
align_offset
int
avoid_collisions
bool
false
collision_padding
Union[float, int, dict]
sticky
"partial" | "always"
hide_when_detached
bool
false
Event Triggers
See the full list of default event triggers
Trigger
Description
on_open_auto_focus
Fired when the dialog is opened.
on_close_auto_focus
Fired when the dialog is closed.
on_escape_key_down
Fired when the escape key is pressed.
on_pointer_down_outside
Fired when the pointer is down outside the dialog.
on_focus_outside
Fired when focus moves outside the dialog.
```

on_interact_outside

Fired when the pointer interacts outside the dialog.

rx.popover.trigger

Wraps the control that will open the popover.

Props

No component specific props

Event Triggers

See the full list of default event triggers

rx.popover.close

Wraps the control that will close the popover.

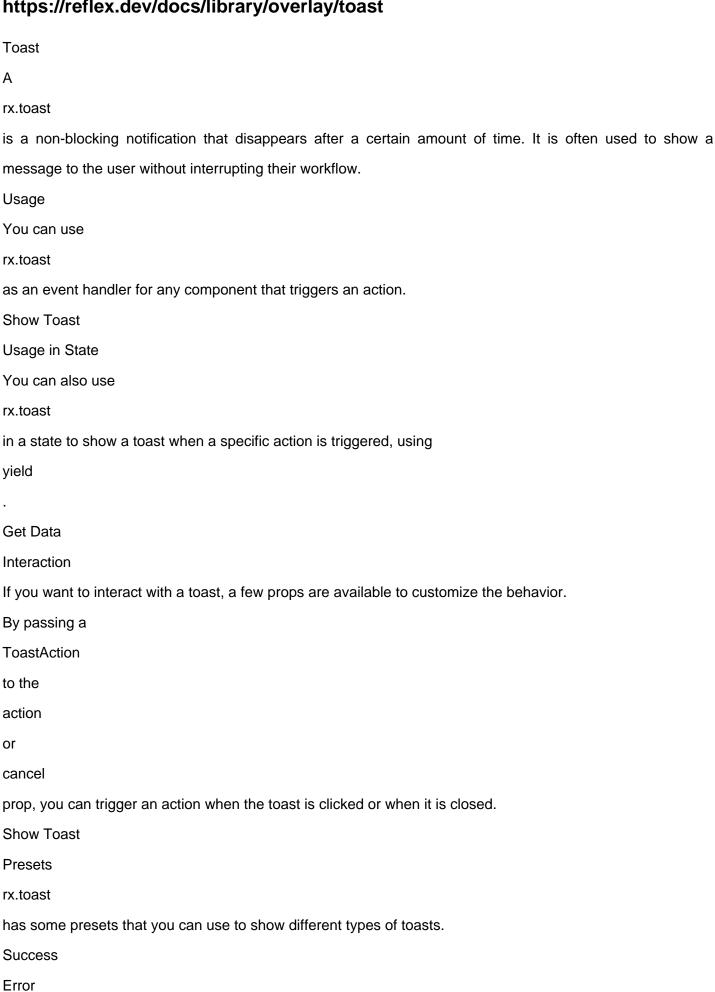
Props

No component specific props

Event Triggers

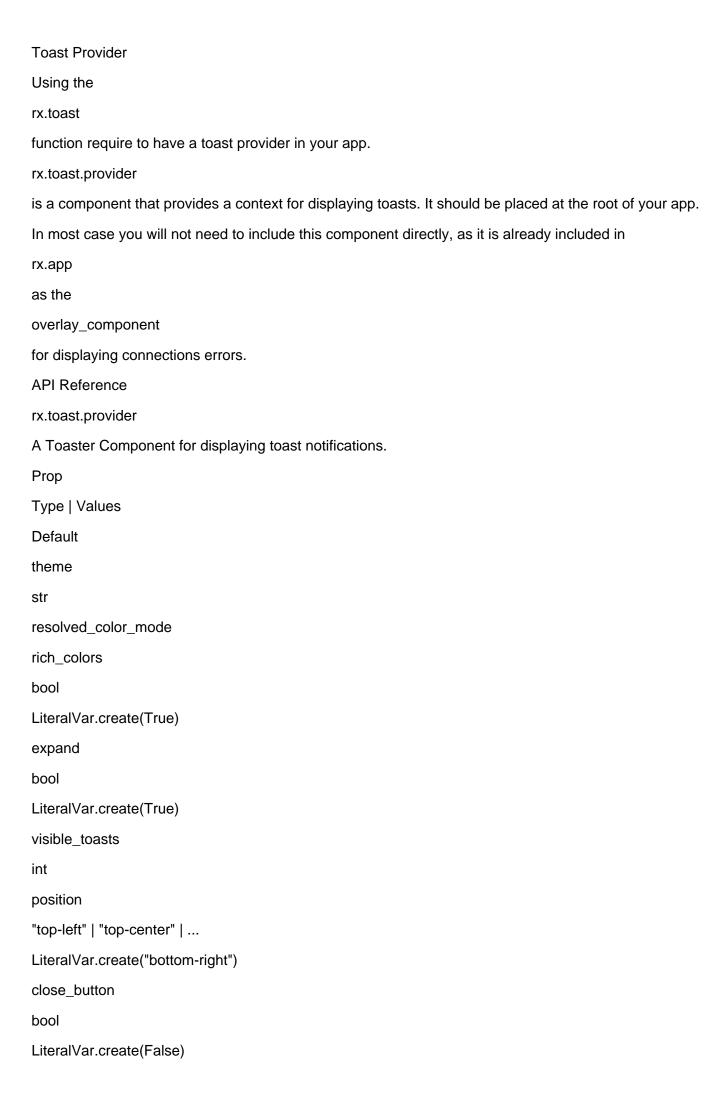
See the full list of default event triggers

https://reflex.dev/docs/library/overlay/toast



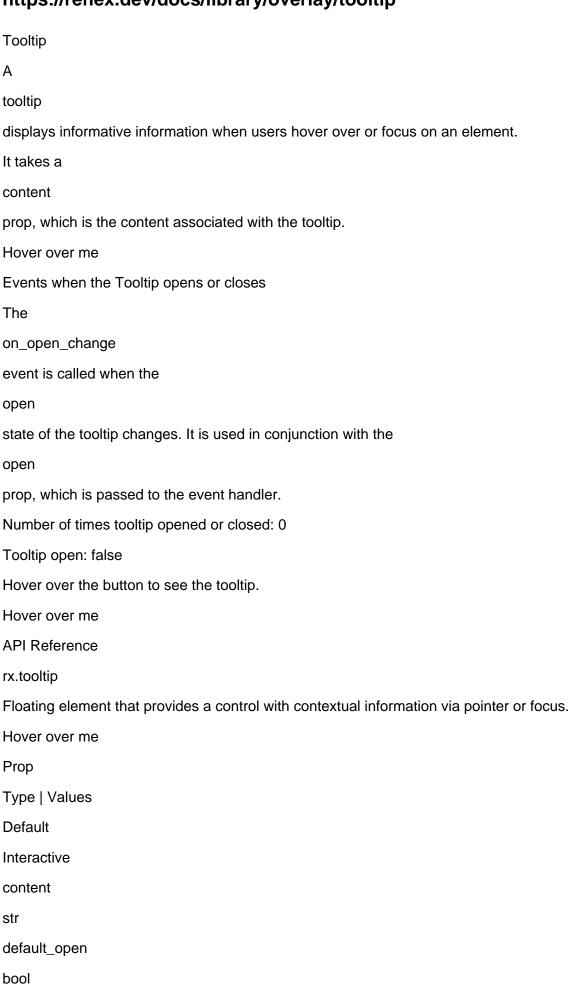
Warning
Info
Customization
If the presets don't fit your needs, you can customize the toasts by passing to
rx.toast
or to
rx.toast.options
some kwargs.
Custom
The following props are available for customization:
description
:
str Var
: Toast's description, renders underneath the title.
close_button
:
bool
: Whether to show the close button.
invert
:
bool
: Dark toast in light mode and vice versa.
important
:
bool
: Control the sensitivity of the toast for screen readers.
duration
:
int
: Time in milliseconds that should elapse before automatically closing the toast.
position
:
LiteralPosition
: Position of the toast.

dismissible
:
bool
: If false, it'll prevent the user from dismissing the toast.
action
:
ToastAction
: Renders a primary button, clicking it will close the toast.
cancel
:
ToastAction
: Renders a secondary button, clicking it will close the toast.
id
;
str Var
: Custom id for the toast.
unstyled
;
bool
: Removes the default styling, which allows for easier customization.
style
:
Style
: Custom style for the toast.
on_dismiss
;
Any
: The function gets called when either the close button is clicked, or the toast is swiped.
on_auto_close
;
Any
: Function that gets called when the toast disappears automatically after it's timeout (
duration
prop).



offset
str
dir
str
hotkey
str
invert
bool
toast_options
ToastProps
gap
int
loading_icon
Icon
pause_when_page_is_hidden
bool
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/overlay/tooltip



open
bool
side
"top" "right"
side_offset
Union[int, float]
align
"start" "center"
align_offset
Union[int, float]
avoid_collisions
bool
false
collision_padding
Union[float, int, dict]
arrow_padding
Union[int, float]
sticky
"partial" "always"
hide_when_detached
bool
false
delay_duration
Union[int, float]
disable_hoverable_content
bool
false
force_mount
bool
false
aria_label
str
Event Triggers
See the full list of default event triggers

Trigger

Description

on_open_change

Fired when the open state changes.

on_escape_key_down

Fired when the escape key is pressed.

on_pointer_down_outside

Fired when the pointer is down outside the tooltip.

https://reflex.dev/docs/library/tables-and-data-grids

Tables And Data Grids

Powerful table components for organizing and displaying data efficiently. Includes versatile options like standard tables, interactive datatables, and editable data grids. Perfect for creating responsive, user-friendly interfaces that present information clearly and allow for easy data manipulation.

Data Editor

Data Table

Table

https://reflex.dev/docs/library/tables-and-data-grids/data-editor

,
Data Editor
A datagrid editor based on
Glide Data Grid
This component is introduced as an alternative to the
datatable
to support editing the displayed data.
Columns
The columns definition should be a
list
of
dict
, each
dict
describing the associated columns.
Property of a column dict:
title
: The text to display in the header of the column.
id
: An id for the column, if not defined, will default to a lower case of
title
width
: The width of the column.
type
: The type of the columns, default to
"str"
Data
The
data
props of
rx.data_editor
accept a

list
of
list
, where each
list
represent a row of data to display in the table.
Simple Example
Here is a basic example of using the data_editor representing data with no interaction and no styling. Below
we define the
columns
and the
data
which are taken in by the
rx.data_editor
component. When we define the
columns
we must define a
title
and a
type
for each column we create. The columns in the
data
must then match the defined
type
or errors will be thrown.
Interactive Example
Here we define a State, as shown below, that allows us to print the location of the cell as a heading when we
click on it, using the
on_cell_clicked
event trigger
. Check out all the other
event triggers
that you can use with datatable at the bottom of this page. We also define a
group

with a label Data . This groups all the columns with this group label under a larger group Data as seen in the table below. Cell clicked: Styling Example Now let's style our datatable to make it look more aesthetic and easier to use. We must first import DataEditorTheme and then we can start setting our style props as seen below in dark_theme We then set these themes using theme=DataEditorTheme(**dark_theme) . On top of the styling we can also set some props to make some other aesthetic changes to our datatable. We have set the row_height to equal 50 so that the content is easier to read. We have also made the smooth_scroll_x and smooth_scroll_y equal True so that we can smoothly scroll along the columns and rows. Finally, we added column_select=single , where column select can take any of the following values none single

```
int
min_column_width
int
row_height
int
row_markers
"none" | "number" | ...
row_marker_start_index
int
row_marker_width
int
smooth_scroll_x
bool
smooth_scroll_y
bool
vertical_border
bool
column_select
"none" | "single" | ...
prevent_diagonal_scrolling
bool
overscroll_x
int
overscroll_y
int
scroll_offset_x
int
scroll_offset_y
int
theme
Union[DataEditorTheme, dict]
Event Triggers
See the full list of default event triggers
Trigger
```

Description on_cell_activated Fired when a cell is activated. on_cell_clicked Fired when a cell is clicked. on_cell_context_menu Fired when a cell is right-clicked. on_cell_edited Fired when a cell is edited. on_group_header_clicked Fired when a group header is clicked. on_group_header_context_menu Fired when a group header is right-clicked. on_group_header_renamed Fired when a group header is renamed. on_header_clicked Fired when a header is clicked. on_header_context_menu Fired when a header is right-clicked. on_header_menu_click Fired when a header menu item is clicked. on_item_hovered Fired when an item is hovered. on_delete Fired when a selection is deleted. on_finished_editing Fired when editing is finished. on_row_appended Fired when a row is appended. on_selection_cleared Fired when the selection is cleared. on_column_resize Fired when a column is resized.

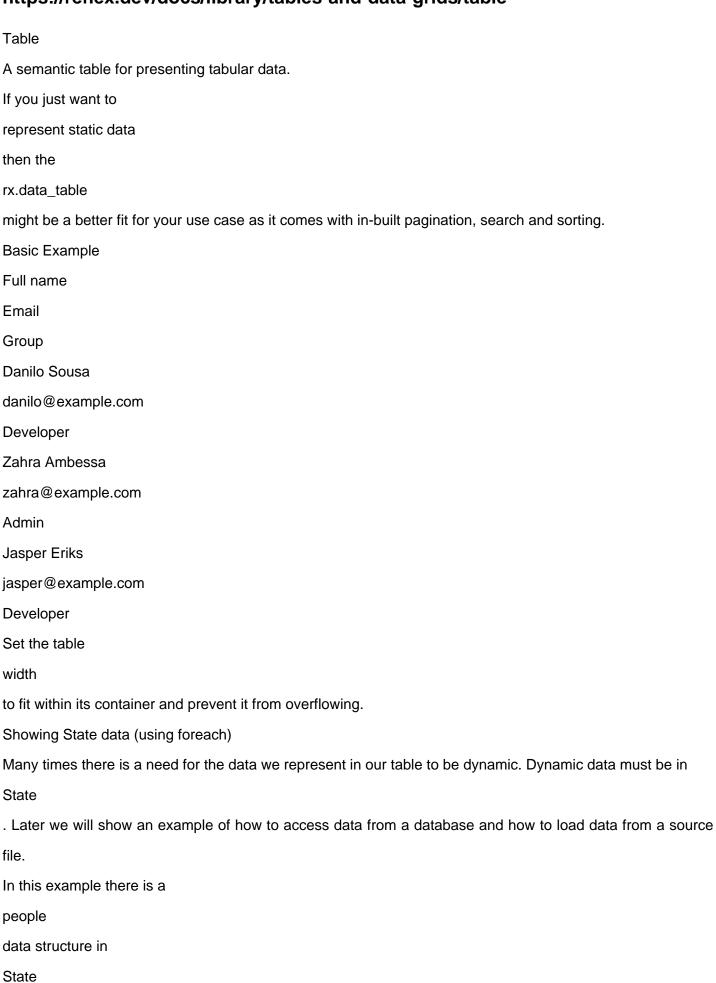
https://reflex.dev/docs/library/tables-and-data-grids/data-table

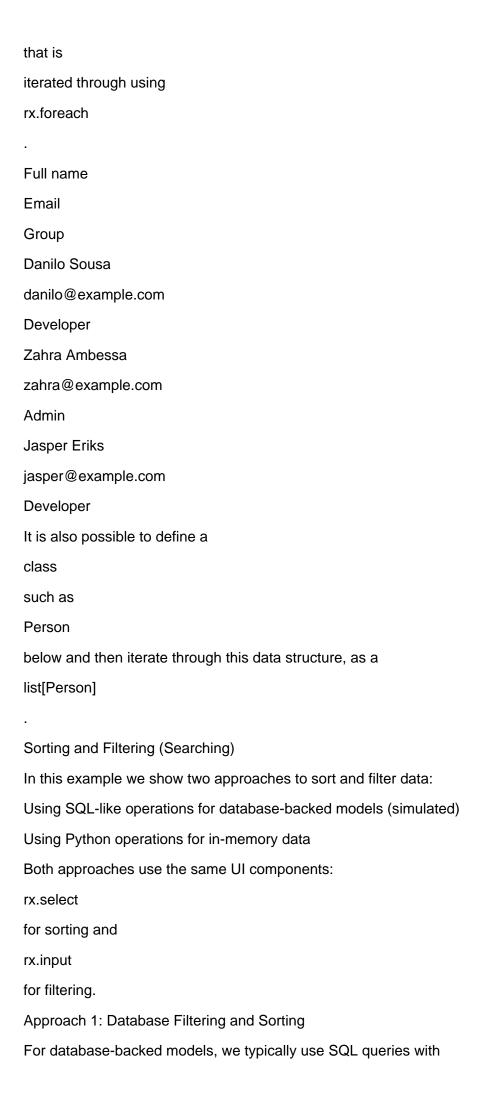
Data Table

See the full list of default event triggers

The data table component is a great way to display static data in a table format. You can pass in a pandas dataframe to the data prop to create the table. In this example we will read data from a csv file, convert it to a pandas dataframe and display it in a data_table. We will also add a search, pagination, sorting to the data_table to make it more accessible. If you want to add, edit or remove data in your app or deal with anything but static data then the rx.table might be a better fit for your use case. The example below shows how to create a data table from from a list. **API Reference** rx.data_table A data table component. Prop Type | Values Default data Any columns Sequence search bool sort bool resizable bool pagination Union[bool, dict] **Event Triggers**

https://reflex.dev/docs/library/tables-and-data-grids/table





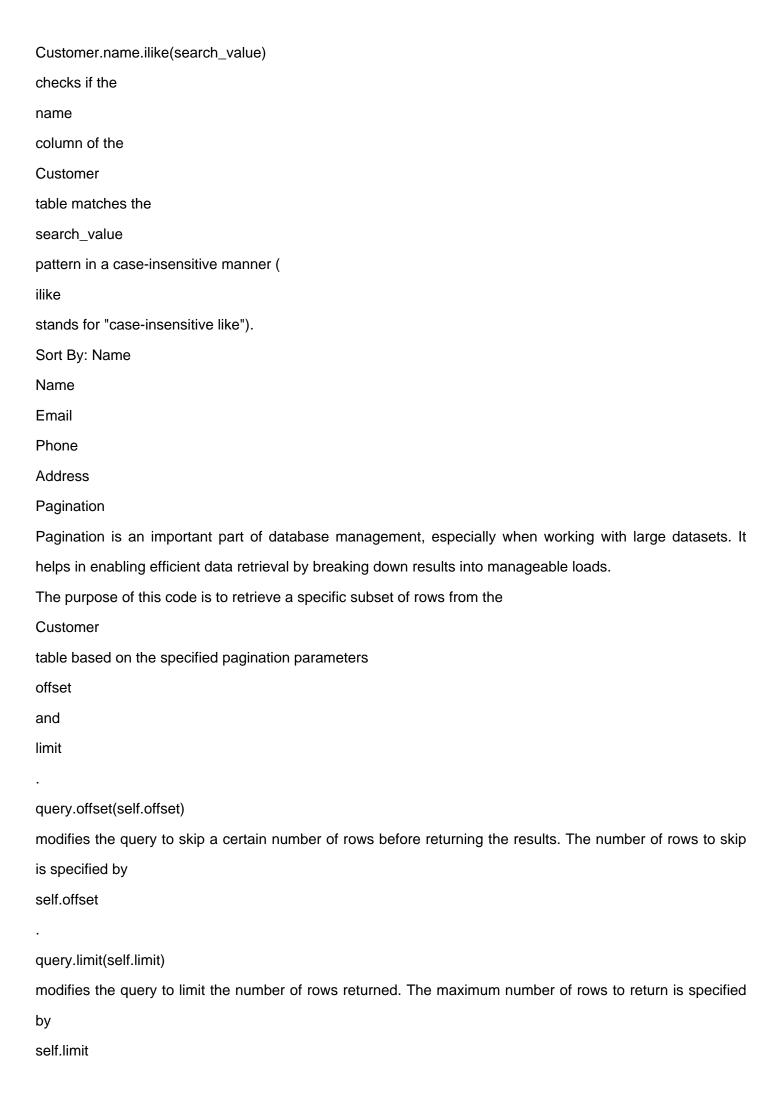
select
,
where
, and
order_by
. In this example, we'll simulate this behavior with mock data.
Sort By: Name
Name
Email
Phone
Address
Approach 2: In-Memory Filtering and Sorting
For in-memory data, we use Python operations like
sorted()
and list comprehensions.
The state variable
_people
is set to be a backend-only variable. This is done in case the variable is very large in order to reduce network
traffic and improve performance.
When a
select
item is selected, the
on_change
event trigger is hooked up to the
set_sort_value
event handler. Every base var has a built-in event handler to set its value for convenience, called
set_VARNAME
current_people
is an
rx.var(cache=True)
. It is a var that is only recomputed when the other state vars it depends on change. This ensures that the
People
shown in the table are always up to date whenever they are searched or sorted.

Sort By: full_name
Full name
Email
Group
Danilo Sousa
danilo@example.com
Developer
Zahra Ambessa
zahra@example.com
Admin
Jasper Eriks
zjasper@example.com
B-Developer
When to Use Each Approach
Database Approach
: Best for large datasets or when the data already exists in a database
In-Memory Approach
: Best for smaller datasets, prototyping, or when the data is static or loaded from an API
Both approaches provide the same user experience with filtering and sorting functionality.
Database
The more common use case for building an
rx.table
is to use data from a database.
The code below shows how to load data from a database and place it in an
rx.table
Loading data into table
A
Customer
model
is defined that inherits from
rx.Model

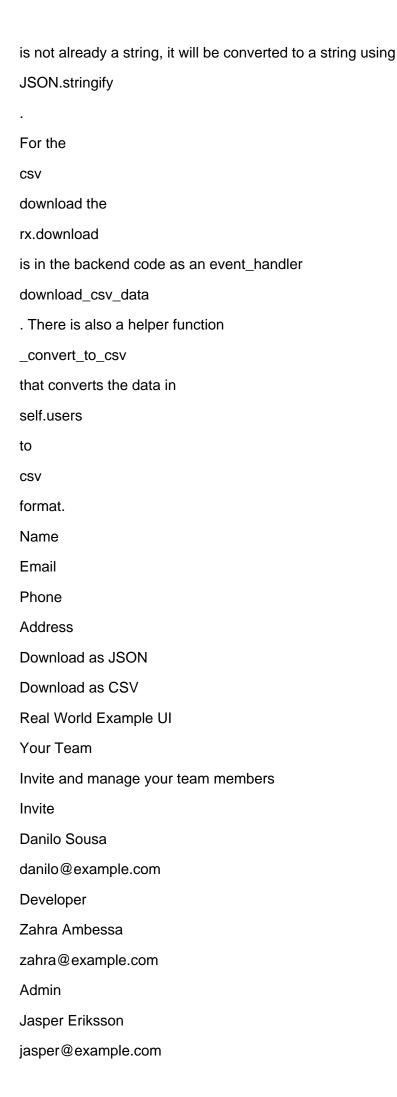
```
load_entries
event handler executes a
query
that is used to request information from a database table. This
load_entries
event handler is called on the
on_mount
event trigger of the
rx.table.root
If you want to load the data when the page in the app loads you can set
on_load
in
app.add_page()
to equal this event handler, like
app.add_page(page_name, on_load=State.load_entries)
Name
Email
Phone
Address
Filtering (Searching) and Sorting
In this example we sort and filter the data.
For sorting the
rx.select
component is used. The data is sorted based on the attributes of the
Customer
class. When a select item is selected, as the
on_change
event trigger is hooked up to the
sort_values
event handler, the data is sorted based on the state variable
sort_value
attribute selected.
```

```
sort_column
based on the state variable
sort value
, it gets the order using the
asc
function from sql and finally uses the
order_by
function.
For filtering the
rx.input
component is used. The data is filtered based on the search query entered into the
rx.input
component. When a search query is entered, as the
on_change
event trigger is hooked up to the
filter_values
event handler, the data is filtered based on if the state variable
search_value
is present in any of the data in that specific
Customer
The
%
character before and after
search_value
makes it a wildcard pattern that matches any sequence of characters before or after the
search value
query.where(...)
modifies the existing query to include a filtering condition. The
or_
operator is a logical OR operator that combines multiple conditions. The query will return results that match
any of these conditions.
```

The sorting query gets the

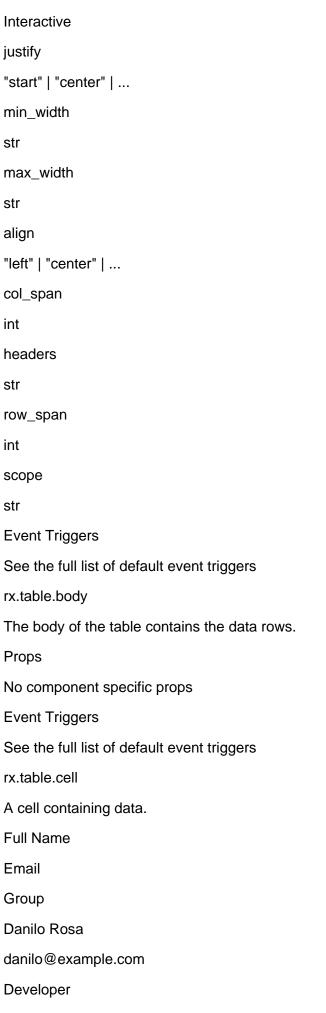


Prev
Page 1 / 0
Next
Name
Email
Phone
Address
More advanced examples
The real power of the
rx.table
comes where you are able to visualise, add and edit data live in your app. Check out these apps and code to
see how this is done: app:
https://customer-data-app.reflex.run
code:
https://github.com/reflex-dev/reflex-examples/tree/main/customer_data_app
and code:
https://github.com/reflex-dev/data-viewer
Download
Most users will want to download their data after they have got the subset that they would like in their table.
In this example there are buttons to download the data as a
son
and as a
CSV
For the
son
download the
rx.download
s in the frontend code attached to the
on_click
event trigger for the button. This works because if the
Var



Developer
API Reference
rx.table.root
A semantic table for presenting tabular data.
Full Name
Email
Group
Danilo Rosa
danilo@example.com
Developer
Zahra Ambessa
zahra@example.com
Admin
Prop
Type Values
Default
Interactive
size
"1" "2"
variant
"surface" "ghost"
align
"left" "center"
summary
str
Event Triggers
See the full list of default event triggers
rx.table.header
The header of the table defines column names and other non-data elements.
Props
No component specific props
Event Triggers
See the full list of default event triggers
rx.table.row

A row containing table cells.
Full Name
Email
Group
Danilo Rosa
danilo@example.com
danilo@yahoo.com
danilo@gmail.com
Developer
Zahra Ambessa
zahra@example.com
Admin
Prop
Type Values
Default
Interactive
align
"start" "center"
Event Triggers
See the full list of default event triggers
rx.table.column_header_cell
A table cell that is semantically treated as a column header.
Full Name
Email
Group
Danilo Rosa
danilo@example.com
Developer
Zahra Ambessa
zahra@example.com
Admin
Prop
Type Values
Default

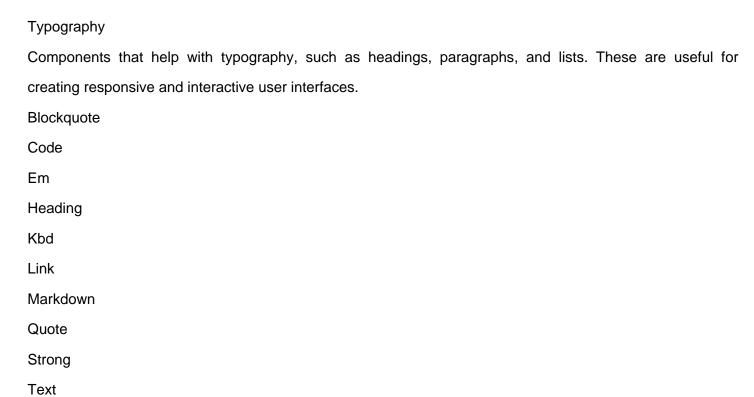


Zahra Ambessa

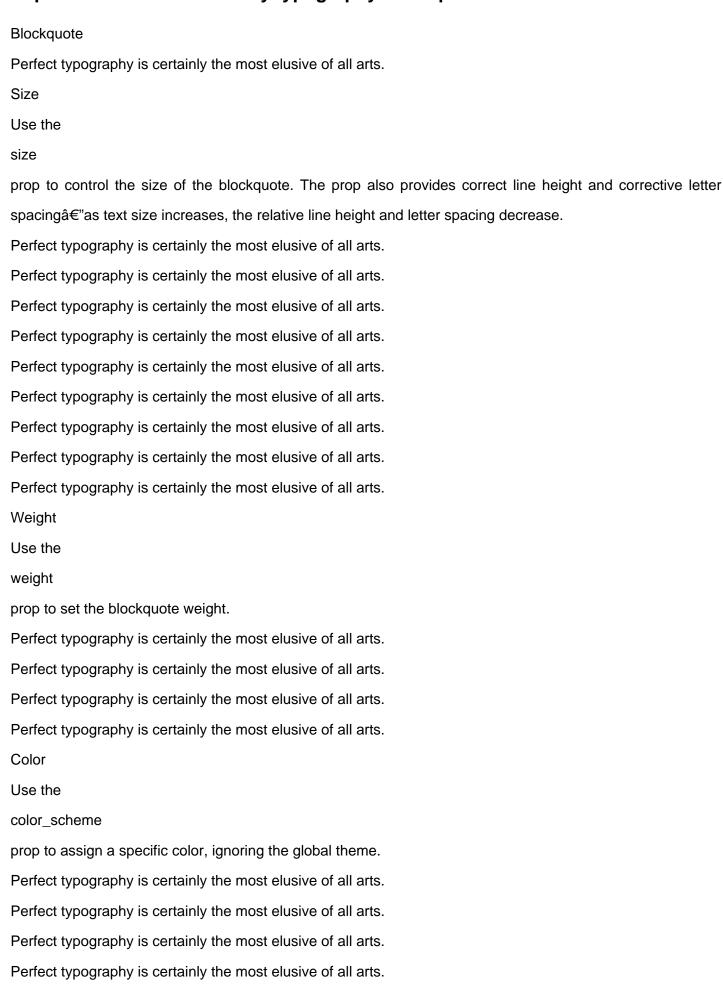
zahra@example.com
Admin
Prop
Type Values
Default
Interactive
justify
"start" "center"
min_width
str
max_width
str
align
"left" "center"
col_span
int
headers
str
row_span
int
Event Triggers
See the full list of default event triggers
rx.table.row_header_cell
A table cell that is semantically treated as a row header.
Full Name
Email
Group
Danilo Rosa
danilo@example.com
Developer
Zahra Ambessa
zahra@example.com
Admin
Prop

Type Values
Default
Interactive
justify
"start" "center"
min_width
str
max_width
str
align
"left" "center"
col_span
int
headers
str
row_span
int
scope
str
Event Triggers
See the full list of default event triggers

https://reflex.dev/docs/library/typography

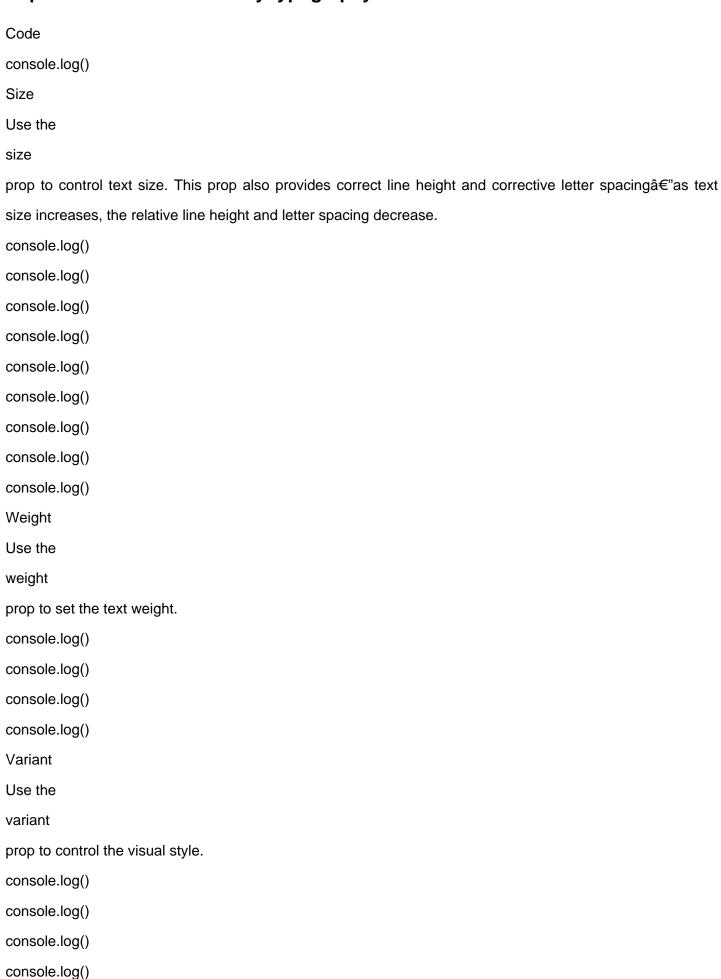


https://reflex.dev/docs/library/typography/blockquote



High Contrast
Use the
high_contrast
prop to increase color contrast with the background.
Perfect typography is certainly the most elusive of all arts.
Perfect typography is certainly the most elusive of all arts.
API Reference
rx.blockquote
A block level extended quotation.
Test
Prop
Type Values
Default
Interactive
size
"1" "2"
weight
"light" "regular"
color_scheme
"tomato" "red"
tomato
high_contrast
high_contrast
high_contrast bool
high_contrast bool false
high_contrast bool false cite
high_contrast bool false cite str

https://reflex.dev/docs/library/typography/code



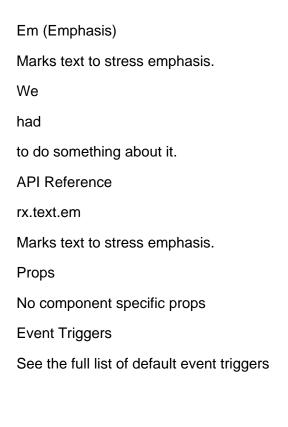
```
Color
Use the
color_scheme
prop to assign a specific color, ignoring the global theme.
console.log()
console.log()
console.log()
console.log()
High Contrast
Use the
high_contrast
prop to increase color contrast with the background.
console.log()
console.log()
console.log()
console.log()
console.log()
console.log()
console.log()
console.log()
API Reference
rx.code
A block level extended quotation.
Test
Prop
Type | Values
Default
Interactive
variant
"classic" | "solid" | ...
size
"1" | "2" | ...
weight
```

"light" | "regular" | ...

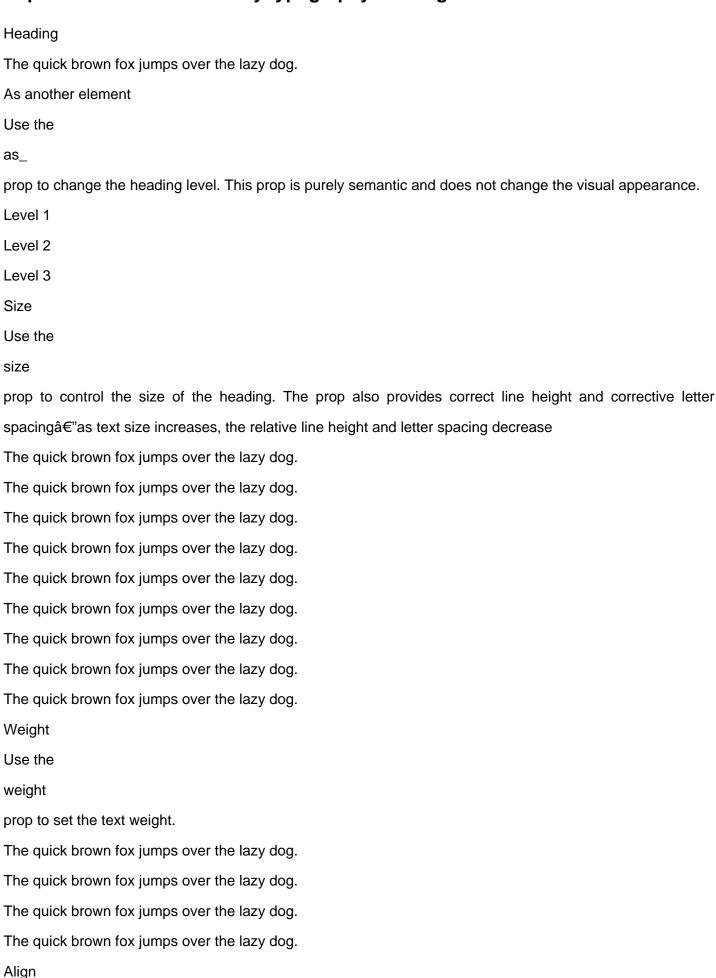
color_scheme
"tomato" "red"
tomato
high_contrast
bool
false
Event Triggers

See the full list of default event triggers

https://reflex.dev/docs/library/typography/em



https://reflex.dev/docs/library/typography/heading

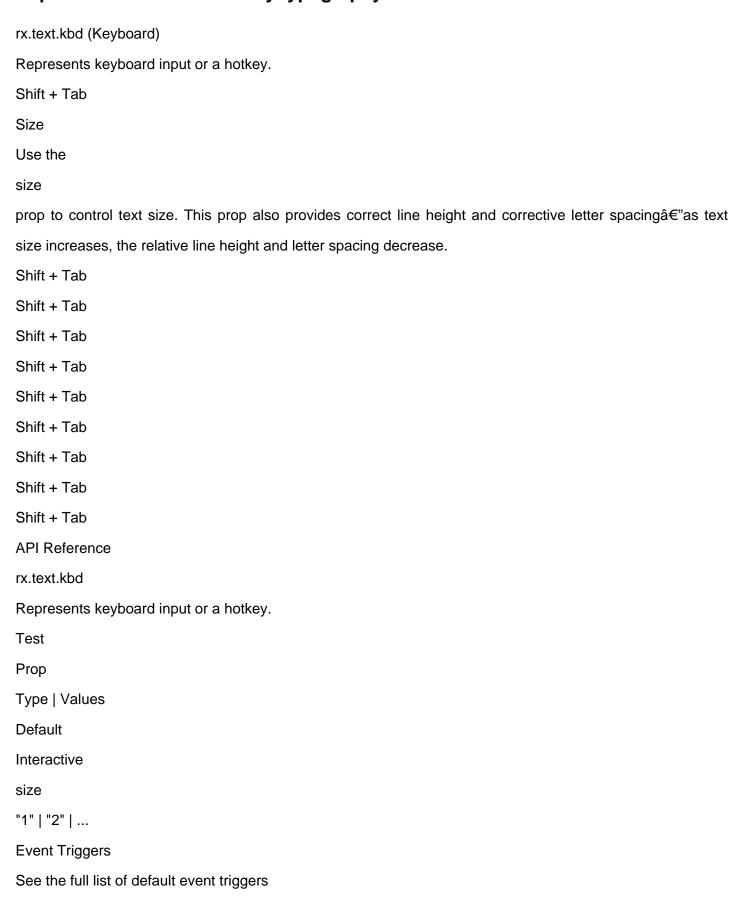


Use the
align
prop to set text alignment.
Left-aligned
Center-aligned
Right-aligned
Trim
Use the
trim
prop to trim the leading space at the start, end, or both sides of the text.
Without Trim
With Trim
Trimming the leading is useful when dialing in vertical spacing in cards or other "boxy― components.
Otherwise, padding looks larger on top and bottom than on the sides.
Without trim
The goal of typography is to relate font size, line height, and line width in a proportional way that maximizes
beauty and makes reading easier and more pleasant.
With trim
The goal of typography is to relate font size, line height, and line width in a proportional way that maximizes
beauty and makes reading easier and more pleasant.
Color
Use the
color_scheme
prop to assign a specific color, ignoring the global theme.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
High Contrast
Use the
high_contrast
prop to increase color contrast with the background.
prop to increase color contrast with the background. The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog. **API Reference** rx.heading A foundational text primitive based on the element. Test Prop Type | Values Default Interactive as_child bool as_ str size "1" | "2" | ... weight "light" | "regular" | ... align "left" | "center" | ... trim "normal" | "start" | ... color_scheme "tomato" | "red" | ... tomato high_contrast bool false **Event Triggers** See the full list of default event triggers

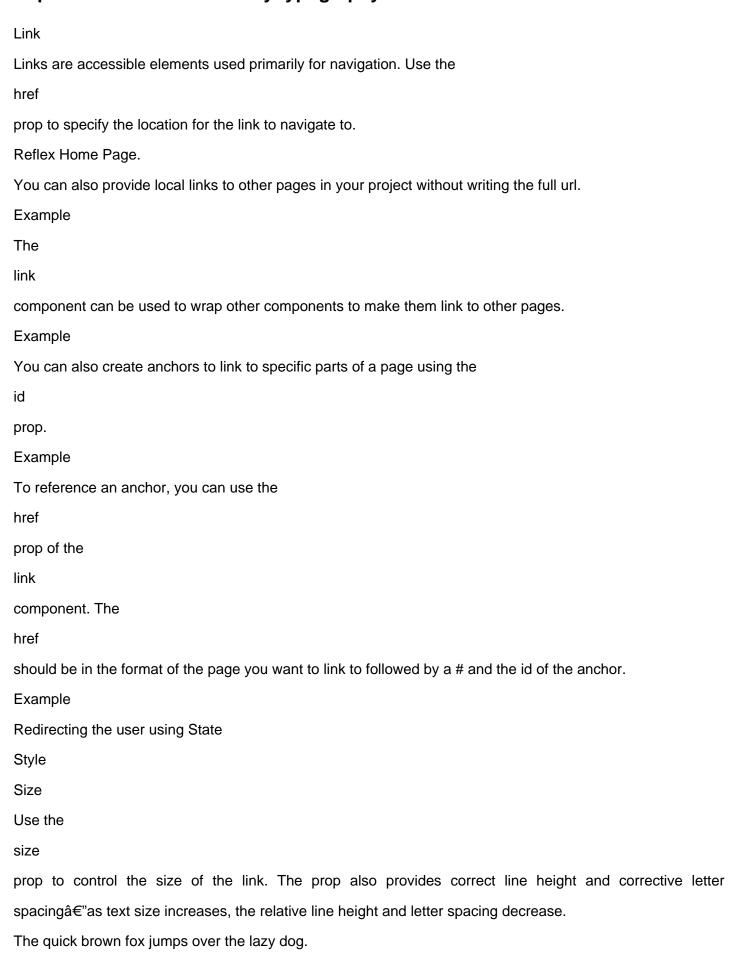
The quick brown fox jumps over the lazy dog.

https://reflex.dev/docs/library/typography/kbd



https://reflex.dev/docs/library/typography/link

The quick brown fox jumps over the lazy dog.



The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. **High Contrast** Use the high_contrast prop to increase color contrast with the background. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. **API Reference** rx.link A semantic element for navigation between pages. Test Prop Type | Values Default Interactive as_child bool size "1" | "2" | ... weight "light" | "regular" | ... trim "normal" | "start" | ... underline "auto" | "hover" | ... color_scheme "tomato" | "red" | ... tomato high_contrast bool false

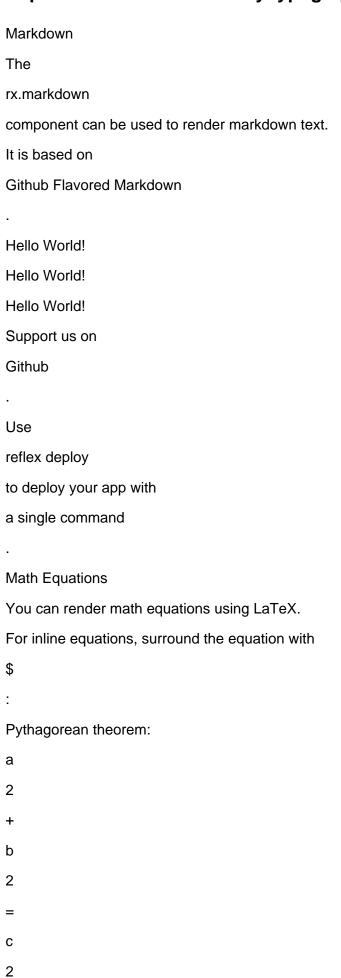
is_external bool

false

Event Triggers

See the full list of default event triggers

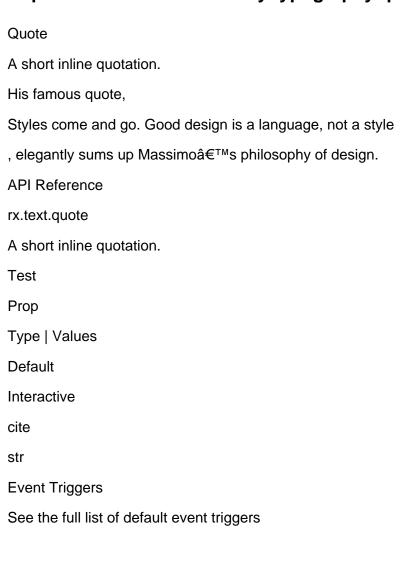
https://reflex.dev/docs/library/typography/markdown



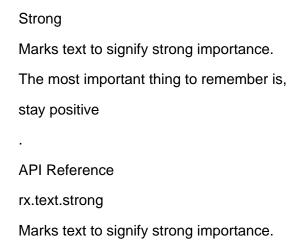
```
a^2 + b^2 = c^2
а
2
+
b
2
=
С
2
Syntax Highlighting
You can render code blocks with syntax highlighting using the ```{language} syntax:
```python
import reflex as rx
from .pages import index
app = rx.App()
app.add_page(index)
...
Tables
You can render tables using the
syntax:
Syntax
Description
Header
Title
Paragraph
Text
Component Map
You can specify which components to use for rendering markdown elements using the
component_map
prop.
Each key in the
component_map
```

prop is a markdown element, and the value is	
a function that takes the text of the element as input and returns a Reflex component.	
Гће	
codeblock	
and	
a	
ags are special cases. In addition to the	
ext	
they also receive a	
props	
argument containing additional props for the component.	
Hello World!	
This is a Subheader	
And Another Subheader	
Here is some	
code	
``python	
mport reflex as rx	
component = rx.text("Hello World!")	
And then some more text here, followed by a link to	
Reflex	
API Reference	
x.markdown	
A markdown component.	
Props	
No component specific props	
Event Triggers	
See the full list of default event triggers	

### https://reflex.dev/docs/library/typography/quote



## https://reflex.dev/docs/library/typography/strong



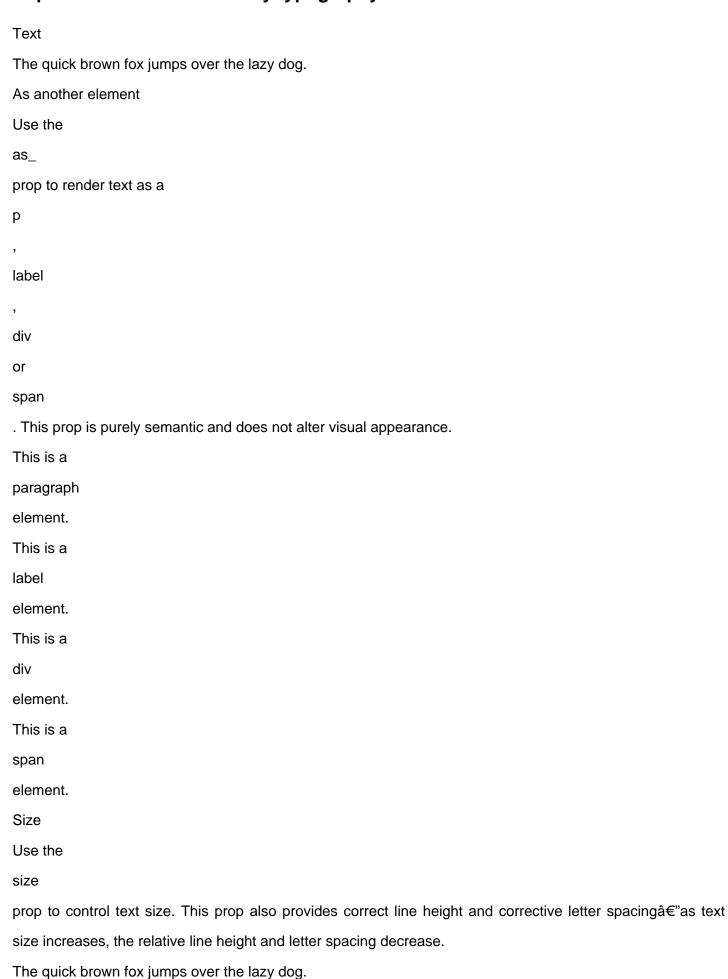
Props

No component specific props

**Event Triggers** 

See the full list of default event triggers

### https://reflex.dev/docs/library/typography/text



The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
Sizes 2â€"4 are designed to work well for long-form content. Sizes 1â€"3 are designed to work well for U
labels.
Weight
Use the
weight
prop to set the text weight.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
Align
Use the
align
prop to set text alignment.
Left-aligned
Center-aligned
Right-aligned
Trim
Use the
trim
prop to trim the leading space at the start, end, or both sides of the text box.
Without Trim
With Trim
Trimming the leading is useful when dialing in vertical spacing in cards or other "boxy― components
Otherwise, padding looks larger on top and bottom than on the sides.

Without trim

The goal of typography is to relate font size, line height, and line width in a proportional way that maximizes beauty and makes reading easier and more pleasant. With trim The goal of typography is to relate font size, line height, and line width in a proportional way that maximizes beauty and makes reading easier and more pleasant. Color Use the color\_scheme prop to assign a specific color, ignoring the global theme. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. **High Contrast** Use the high\_contrast prop to increase color contrast with the background. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. With formatting Compose Text with formatting components to add emphasis and structure to content. Look, such a helpful link , an italic emphasis a piece of computer code , and even a hotkey combination ⇧âŒ~A

within the text.

Preformmatting By Default, the browser renders multiple white spaces into one. To preserve whitespace, use the white\_space = "pre" css prop. This is not pre formatted This is pre formatted With form controls Composing text with a form control like checkbox radiogroup , or switch automatically centers the control with the first line of text, even when the text is multi-line. I understand that these documents are confidential and cannot be shared with a third party. **API Reference** rx.text A foundational text primitive based on the element. Test Prop Type | Values Default Interactive as child bool as\_ "p" | "label" | ... Var.create("p") size "1" | "2" | ...

weight

```
"light" | "regular" | ...
align
"left" | "center" | ...
trim
"normal" | "start" | ...
color_scheme
"tomato" | "red" | ...
tomato
high_contrast
bool
false
Event Triggers
See the full list of default event triggers
rx.text.em
Marks text to stress emphasis.
Props
No component specific props
Event Triggers
See the full list of default event triggers
```

#### https://reflex.dev/docs/pages/dynamic-routing

Dynamic Routes

Dynamic routes in Reflex allow you to handle varying URL structures, enabling you to create flexible and adaptable web applications. This section covers regular dynamic routes, catch-all routes, and optional catch-all routes, each with detailed examples.

Regular Dynamic Routes

Regular dynamic routes in Reflex allow you to match specific segments in a URL dynamically. A regular dynamic route is defined by square brackets in a route string / url pattern. For example

/users/[id]

or

/products/[category]

. These dynamic route arguments can be accessed through a state var. For the examples above they would

be

rx.State.id

and

rx.State.category

respectively.

Why is the state var accessed as

rx.State.id

?

Example:

The [pid] part in the route is a dynamic segment, meaning it can match any value provided in the URL. For instance,

/post/5

/post/10

, or

/post/abc

would all match this route.

If a user navigates to

/post/5

State.post\_id

```
will return
, and the page will display
5
as the heading. If the URL is
/post/xyz
, it will display
XYZ
. If the URL is
/post/
without any additional parameter, it will display
Adding Dynamic Routes
Adding dynamic routes uses the
add_page
method like any other page. The only difference is that the route string contains dynamic segments enclosed
in square brackets.
If you are using the
app.add_page
method to define pages, it is necessary to add the dynamic routes first, especially if they use the same
function as a non dynamic route.
For example the code snippet below will:
But if we switch the order of adding the pages, like in the example below, it will not work:
Catch-All Routes
Catch-all routes in Reflex allow you to match any number of segments in a URL dynamically.
Example:
In this case, the
...splat
catch-all pattern captures any number of segments after
/users/
, allowing URLs like
/users/2/posts/john/
and
```

/users/1/posts/john/doe/
to match the route.
Catch-all routes must be named
splat
and be placed at the end of the URL pattern to ensure proper route matching.
Routes Validation Table
Route Pattern
Example URI
valid
/users/posts
/users/posts
valid
/products/[category]
/products/electronics
valid
/users/[username]/posts/[id]
/users/john/posts/5
valid
/users/[[splat]]/posts
/users/john/posts
invalid
/users/john/doe/posts
invalid
/users/[[splat]]
/users/john/
valid
/users/john/doe
valid
/products/[category]/[[splat]]
/products/electronics/laptops
valid
/products/electronics/laptops/lenovo
valid
/products/[category]/[[splat]]

/products/electronics
valid
/products/electronics/laptops
valid
/products/electronics/laptops/lenovo
valid
/products/electronics/laptops/lenovo/thinkpad
valid
/products/[category]/[[splat]]/[[splat]]
/products/electronics/laptops
invalid
/products/electronics/laptops/lenovo
invalid
/products/electronics/laptops/lenovo/thinkpad
invalid

#### https://reflex.dev/docs/pages/overview

Pages

Pages map components to different URLs in your app. This section covers creating pages, handling URL arguments, accessing query parameters, managing page metadata, and handling page load events.

Adding a Page

You can create a page by defining a function that returns a component.

By default, the function name will be used as the route, but you can also specify a route.

In this example we create three pages:

index

- The root route, available at

•

about

- available at

/about

custom

- available at

/custom-route

Index is a special exception where it is available at both

/

and

/index

. All other pages are only available at their specified route.

Video: Pages and URL Routes

Page Decorator

You can also use the

@rx.page

decorator to add a page.

This is equivalent to calling

app.add\_page

with the same arguments.

Remember to import the modules defining your decorated pages.

**Navigating Between Pages** 

Links

Links
are accessible elements used primarily for navigation. Use the
href
prop to specify the location for the link to navigate to.
Reflex Home Page.
You can also provide local links to other pages in your project without writing the full url.
Example
To open the link in a new tab, set the
is_external
prop to
True
•
Open in new tab
Check out the
link docs
to learn more.
Video: Link-based Navigation
Redirect
Redirect the user to a new path within the application using
rx.redirect()
•
path
: The destination path or URL to which the user should be redirected.
external
: If set to True, the redirection will open in a new tab. Defaults to
False
•
open in tab
open in new tab
Redirect can also be run from an event handler in State, meaning logic can be added behind it. It is
necessary to
return
the
rx.redirect()

https://github.com/reflex-dev/reflex/ Change redirect location Redirect to new page in State Video: Redirecting to a New Page **Nested Routes** Pages can also have nested routes. This component will be available at /nested/page Page Metadata You can add page metadata such as: The title to be shown in the browser tab The description as shown in search results The preview image to be shown when the page is shared on social media Any additional metadata Getting the Current Page You can access the current page from the router attribute in any state. See the router docs for all available attributes. The router.page.path attribute allows you to obtain the path of the current page from the router data, for dynamic pages this will contain the slug rather than the actual value used to load the page. To get the actual URL displayed in the browser, use router.page.raw\_path . This will contain all query parameters and dynamic path segments. In the above example, current\_page\_route

```
will contain the route pattern (e.g., /posts/[id]
), while
current_page_url
will contain the actual URL (e.g., /posts/123
).
To get the full URL, access the same attributes with full_
prefix.
Example:
In this example, running on localhost
should display
http://localhost:3000/posts/123/
```

#### https://reflex.dev/docs/recipes



https://reflex.dev/docs/recipes/auth/login-form Login Form The login form is a common component in web applications. It allows users to authenticate themselves and access their accounts. This recipe provides examples of login forms with different elements, such as third-party authentication providers. Default UI UI Code Code Sign in to your account Email address Password Forgot password? Sign in New here? Sign up Icons UI UI Code Code Sign in to your account

Email address

Forgot password?

Password

Sign in

Sign up

UI

UI

Code

New here?

Third-party auth

Code
Sign in to your account
New here?
Sign up
Email address
Password
Forgot password?
Sign in
Or continue with
Sign in with Github
Multiple third-party auth
UI
UI
Code
Code
Sign in to your account
New here?
Sign up
Email address
Password
Forgot password?
Sign in
Or continue with

https://reflex.dev/docs/recipes/auth/signup-form Sign up Form The sign up form is a common component in web applications. It allows users to create an account and access the application's features. This page provides a few examples of sign up forms that you can use in your application. Default UI UI Code Code Create an account Email address Password Agree to Terms and Conditions Register Already registered? Sign in Icons UI UI Code Code Create an account

Email address

Already registered?

Third-party auth

Agree to Terms and Conditions

Password

Register

Sign in

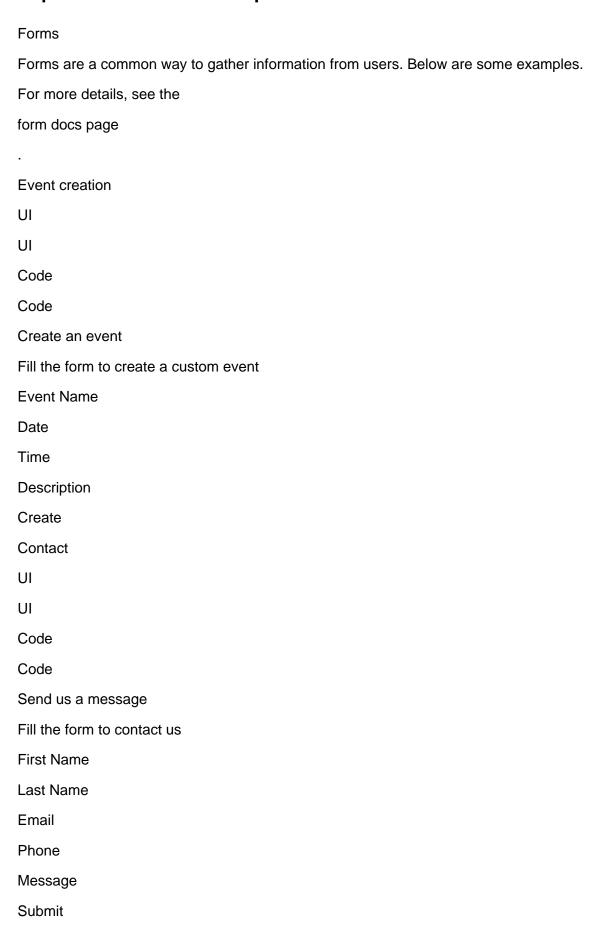
UI

UI

Code

Code
Create an account
Already registered?
Sign in
Email address
Password
Agree to Terms and Conditions
Register
Or continue with
Sign in with Github
Multiple third-party auth
UI
UI
Code
Code
Create an account
Already registered?
Sign in
Email address
Password
Agree to Terms and Conditions
Register
Or continue with

### https://reflex.dev/docs/recipes/content/forms



## https://reflex.dev/docs/recipes/content/grid

Grid
A simple responsive grid layout. We specify the number of columns to the
grid_template_columns
property as a list. The grid will automatically adjust the number of columns based on the screen size.
For details, see the
responsive docs page
Cards
Card 1
Card 2
Card 3
Card 4
Card 5
Card 6
Card 7
Card 8
Card 9
Card 10
Card 11
Card 12
Inset cards
Card 1
Card 2
Card 3
Card 4
Card 5
Card 6
Card 7
Card 8
Card 9
Card 10
Card 11

### https://reflex.dev/docs/recipes/content/multi-column-row

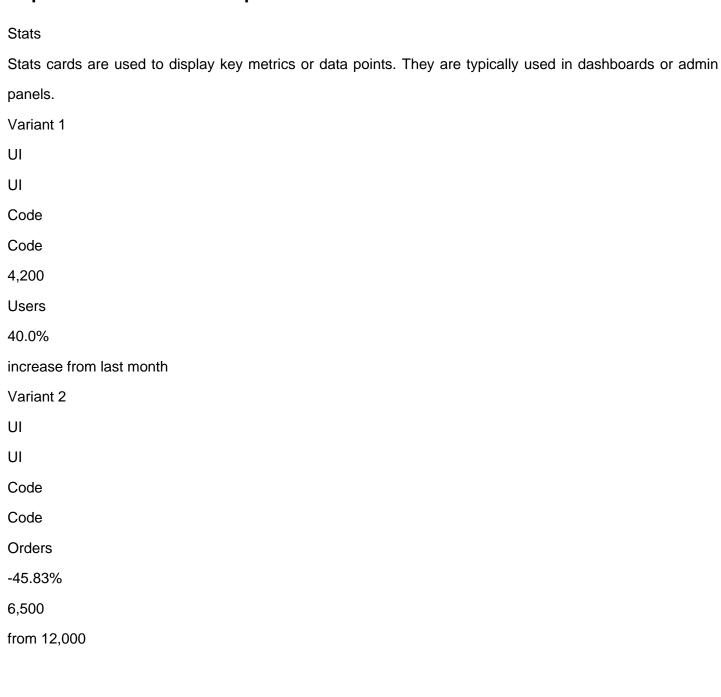
Multi-column and row layout

A simple responsive multi-column and row layout. We specify the number of columns/rows to the flex\_direction property as a list. The layout will automatically adjust the number of columns/rows based on the screen size. For details, see the responsive docs page

Column

Row

## https://reflex.dev/docs/recipes/content/stats



## https://reflex.dev/docs/recipes/content/top-banner

Join our newsletter

Top Banner
Top banners are used to highlight important information or features at the top of a page. They are typically
designed to grab the user's attention and can be used for announcements, navigation, or key messages.
Basic
UI
UI
Code
Code
ReflexCon 2024 -
Join us at the event!
Sign up
UI
UI
Code
Code
Web apps in pure Python. Deploy with a single command.
Sign up
Gradient
UI
UI
Code
Code
The new Reflex version is now available!
Read the release notes
Newsletter
UI
UI
Code
Code

### https://reflex.dev/docs/recipes/layout/footer

Web Design

Web Development

Footer Bar A footer bar is a common UI element located at the bottom of a webpage. It typically contains information about the website, such as contact details and links to other pages or sections of the site. **Basic** UI UI Code Code Reflex © 2024 Reflex, Inc **PRODUCTS** Web Design Web Development E-commerce **Content Management** Mobile Apps **RESOURCES** Blog Case Studies Whitepapers Webinars E-books **Privacy Policy** Terms of Service Newsletter form UI UI Code Code **PRODUCTS** 

E-commerce
Content Management
Mobile Apps
RESOURCES
Blog
Case Studies
Whitepapers
Webinars
E-books
JOIN OUR NEWSLETTER
© 2024 Reflex, Inc
Three columns
UI
UI
Code
Code
PRODUCTS
Web Design
Web Development
E-commerce
Content Management
Mobile Apps
RESOURCES
Blog
Case Studies
Whitepapers
Webinars
E-books
ABOUT US
Our Team
Careers
Contact Us
Privacy Policy
Terms of Service



#### https://reflex.dev/docs/recipes/layout/navbar

Navigation Bar

A navigation bar, also known as a navbar, is a common UI element found at the top of a webpage or application.

It typically provides links or buttons to the main sections of a website or application, allowing users to easily navigate and access the different pages.

Navigation bars are useful for web apps because they provide a consistent and intuitive way for users to navigate through the app.

Having a clear and consistent navigation structure can greatly improve the user experience by making it easy

for users to find the information they need and access the different features of the app.
Video: Example of Using the Navbar Recipe
Basic





About	
Pricing	
Contact	
Reflex	

Dropdown	
UI	

UI	
Code	
Code	

Reflex	
Home	
Services	

Pricing
Contact

Reflex

Search bar
UI
UI
Code
Code
Reflex
Reflex
Icons
UI
UI
Code
Code
Reflex
Home
Pricing
Contact
Services
Reflex
Buttons
UI
UI
Code
Code
Reflex
Home
About
Pricing
Contact
Sign Up
Log In
Reflex
User profile
UI
UI

Code

Code

Reflex

Home

About

Pricing

Contact

Reflex

### https://reflex.dev/docs/recipes/layout/sidebar



My account

user@reflex.dev

Dashboard

Projects

Analytics

Messages

Help & Support

### https://reflex.dev/docs/recipes/others/checkboxes

Smart Checkboxes Group

A smart checkboxes group where you can track all checked boxes, as well as place a limit on how many checks are possible.

Recipe

This recipe use a

dict[str, bool]

for the checkboxes state tracking.

Additionally, the limit that prevent the user from checking more boxes than allowed with a computed var.

## https://reflex.dev/docs/recipes/others/chips

Chips
Chips are compact elements that represent small pieces of information, such as tags or categories. They are
commonly used to select multiple items from a list or to filter content.
Status
UI
UI
Code
Code
Info
Success
Warning
Error
Single selection
UI
UI
Code
Code
Select your reservation time:
2:00
3:00
4:00
5:00
Multiple selection
This example demonstrates selecting multiple skills from a list. It includes buttons to add all skills, clear
selected skills, and select a random number of skills.
UI
UI
Code
Code
Skills (3)
Add All
Clear All

Networking
Security
Cloud
DevOps
Data Science
Al
ML
Robotics
Cybersecurity

Data Management

## https://reflex.dev/docs/recipes/others/dark-mode-toggle

Dark Mode Toggle
The Dark Mode Toggle component lets users switch between light and dark themes
UI
UI
Code
Code

## https://reflex.dev/docs/recipes/others/pricing-cards

\$69.99

Pricing Cards
A pricing card shows the price of a product or service. It typically includes a title, description, price, features,
and a purchase button.
Basic
UI
UI
Code
Code
Beginner
Ideal choice for personal use & for your next project.
\$39
/month
24/7 customer support
Daily backups
Advanced analytics
Customizable templates
Priority email support
Get started
Comparison cards
UI
UI
Code
Code
\$14.99
\$3.99
40 Image Credits
40 credits for image generation
Credits never expire
High quality images
Commercial license
Purchase

\$18.99

**POPULAR** 

250 Image Credits

250 credits for image generation

+30% Extra free credits

Credits never expire

High quality images

Commercial license

Purchase

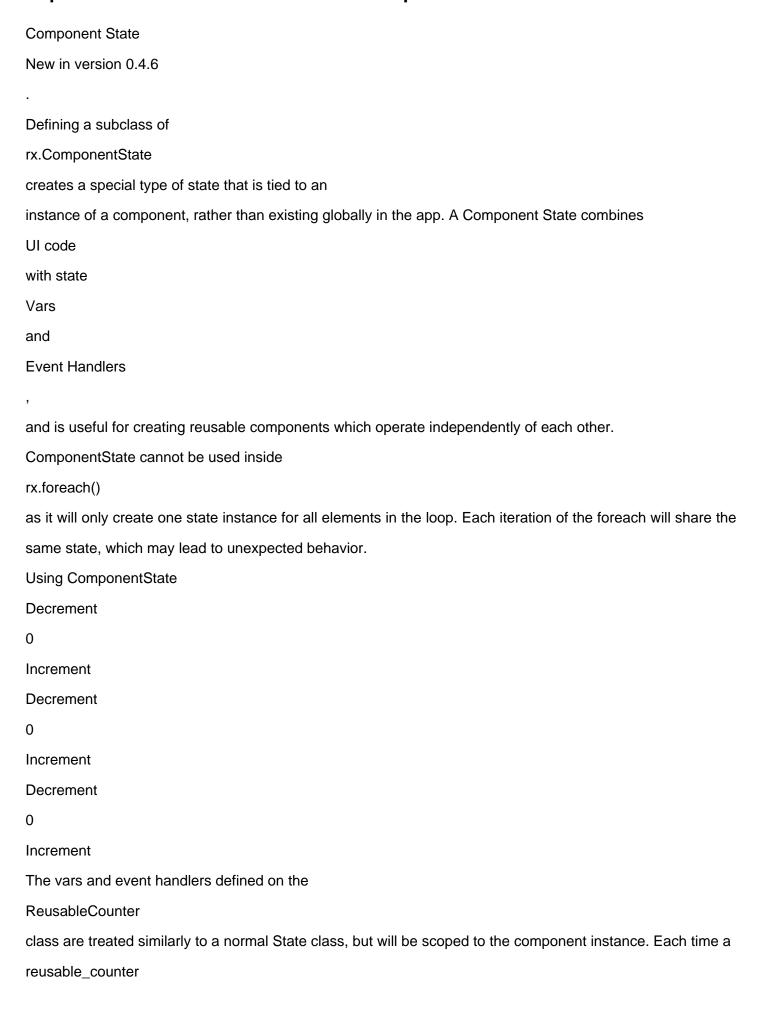
# https://reflex.dev/docs/recipes/others/speed-dial

Code

Code

Speed Dial
A speed dial is a component that allows users to quickly access frequently used actions or pages. It is often
used in the bottom right corner of the screen.
Vertical
UI
UI
Code
Code
Horizontal
UI
UI
Code
Code
Vertical with text
UI
UI
Code
Code
Reveal animation
UI
UI
Code
Code
Menu
UI
111

#### https://reflex.dev/docs/state-structure/component-state



is created, a new state class for that instance of the component is also created.
The
get_component
classmethod is used to define the UI for the component and link it up to the State, which
is accessed via the
cls
argument. Other states may also be referenced by the returned component, but
cls
will always be the instance of the
ComponentState
that is unique to the component being returned.
Passing Props
Similar to a normal Component, the
ComponentState.create
classmethod accepts the arbitrary
*children
and
**props
arguments, and by default passes them to your
get_component
classmethod.
These arguments may be used to customize the component, either by applying defaults or
passing props to certain subcomponents.
In the following example, we implement an editable text component that allows the user to click on
the text to turn it into an input field. If the user does not provide their own
value
or
on_change
props, then the defaults defined in the
EditableText
class will be used.
Reflex is fun
Reflex is fun
Reflex is fun

Because this
EditableText
component is designed to be reusable, it can handle the case
where the
value
and
on_change
are linked to a normal global state.
Global state text
GLOBAL STATE TEXT
Accessing the State
The underlying state class of a
ComponentState
s accessible via the
State
attribute. To use it,
assign an instance of the component to a local variable, then include that instance in the page.
Total: 0
Decrement
0
Increment
Decrement
0
Increment
Other components can also affect a
ComponentState
by referencing its event handlers or vars
via the
State
attribute.
Decrement
Increment
Double

#### https://reflex.dev/docs/state-structure/overview

Substates

Substates allow you to break up your state into multiple classes to make it more manageable. This is useful as your app

grows, as it allows you to think about each page as a separate entity. Substates also allow you to share common state

resources, such as variables or event handlers.

When a particular state class becomes too large, breaking it up into several substates can bring performance benefits by only loading parts of the state that are used to handle a certain event.

Multiple States

One common pattern is to create a substate for each page in your app.

This allows you to think about each page as a separate entity, and makes it easier to manage your code as your app grows.

To create a substate, simply inherit from

rx.State

multiple times:

Separating the states is purely a matter of organization. You can still access the state from other pages by importing the state class.

**Accessing Arbitrary States** 

An event handler in a particular state can access and modify vars in another state instance by calling

the

get\_state

async method and passing the desired state class. If the requested state is not already loaded,

it will be loaded and deserialized on demand.

In the following example, the

GreeterState

accesses the

SettingsState

to get the

salutation

and uses it

to update the

message

var.
Notably, the widget that sets the salutation does NOT have to load the
GreeterState
when handling the
input
on_change
event, which improves performance.
Submit
Accessing Individual Var Values
In addition to accessing entire state instances with
get_state
, you can retrieve individual variable values using the
get_var_value
method:
This async method is particularly useful when you only need a specific value rather than loading the entire
state. Using
get_var_value
can be more efficient than
get_state
when:
You only need to access a single variable from another state
The other state contains a large amount of data
You want to avoid loading unnecessary data into memory
Here's an example that demonstrates how to use
get_var_value
to access data between states:
Get Var Value Example
Get Count Value
Increment
In this example:
We have two separate states:
CounterState
which manages a counter, and
DisplayState

which displays information When you click "Increment", it calls CounterState.increment() to increase the counter value When you click "Show Count", it calls DisplayState.show\_count() which uses get\_var\_value to retrieve just the count value from CounterState without loading the entire state The current count is then displayed in the message This pattern is useful when you have multiple states that need to interact with each other but don't need to access all of each other's data. If the var is not retrievable. get\_var\_value will raise an UnretrievableVarValueError Performance Implications When an event handler is called, Reflex will load the data not only for the substate containing the event handler, but also all of its substates and parent states as well. If a state has a large number of substates or contains a large amount of data, it can slow down processing of events associated with that state. For optimal performance, keep a flat structure with most substate classes directly inheriting from rx.State Only inherit from another state when the parent holds data that is commonly used by the substate. Implementing different parts of the app with separate, unconnected states ensures that only the necessary data is loaded for processing events for a particular page or component. Avoid defining computed vars inside a state that contains a large amount of data, as states with computed vars are always loaded to ensure the values are recalculated.

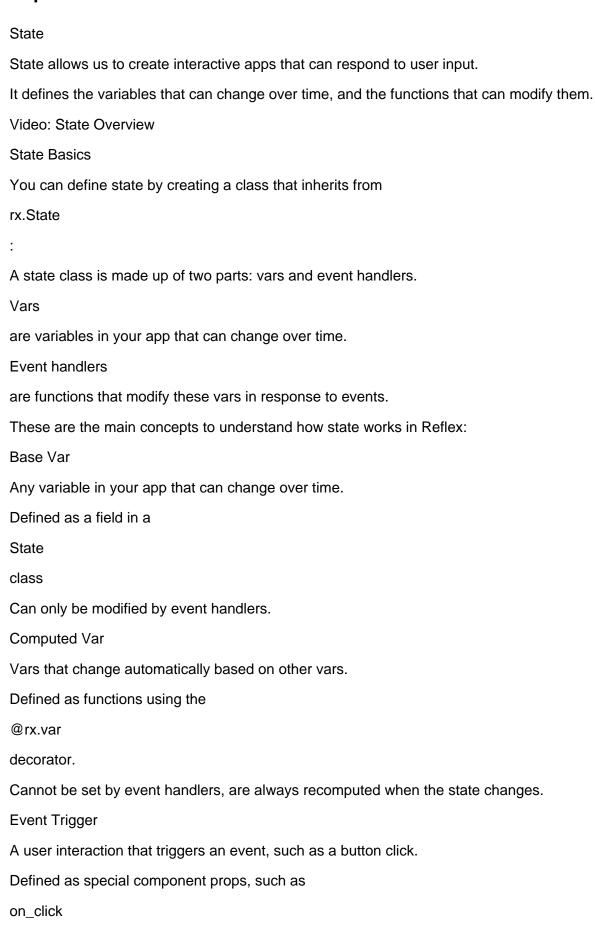
When using computed vars, it better to define them in a state that directly inherits from

rx.State

#### and

does not have other states inheriting from it, to avoid loading unnecessary data.

# https://reflex.dev/docs/state/overview



Can be used to trigger event handlers.

**Event Handlers** Functions that update the state in response to events. Defined as methods in the State class. Can be called by event triggers, or by other event handlers. Example Here is a example of how to use state within a Reflex app. Click the text to change its color. Welcome to Reflex! The base vars are colors and index . They are the only vars in the app that may be directly modified within event handlers. There is a single computed var, color , that is a function of the base vars. It will be computed automatically whenever the base vars change. The heading component links its on\_click event to the ExampleState.next\_color event handler, which increments the color index. With Reflex, you never have to write an API. State vs. Instance? Cannot print a State var. Client States Each user who opens your app has a unique ID and their own copy of the state. This means that each user can interact with the app and modify the state independently of other users. Because Reflex internally creates a new instance of the state for each user, your code should never directly initialize a state class.

Try opening an app in multiple tabs to see how the state changes independently. All user state is stored on the server, and all event handlers are executed on the server. Reflex uses websockets to send events to the server, and to send state updates back to the client.

Helper Methods

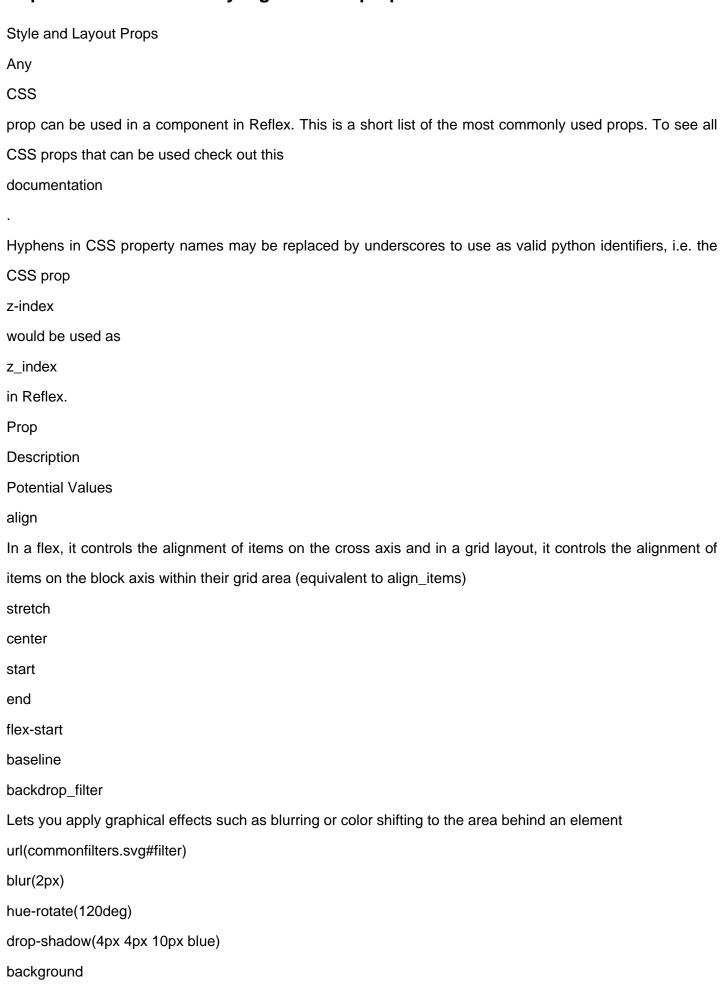
Similar to backend vars, any method defined in a State class that begins with an underscore

-

is considered a helper method. Such methods are not usable as event triggers, but may be called from other event handler methods within the state.

Functionality that should only be available on the backend, such as an authenticated action, should use helper methods to ensure it is not accidentally or maliciously triggered by the client.

# https://reflex.dev/docs/styling/common-props



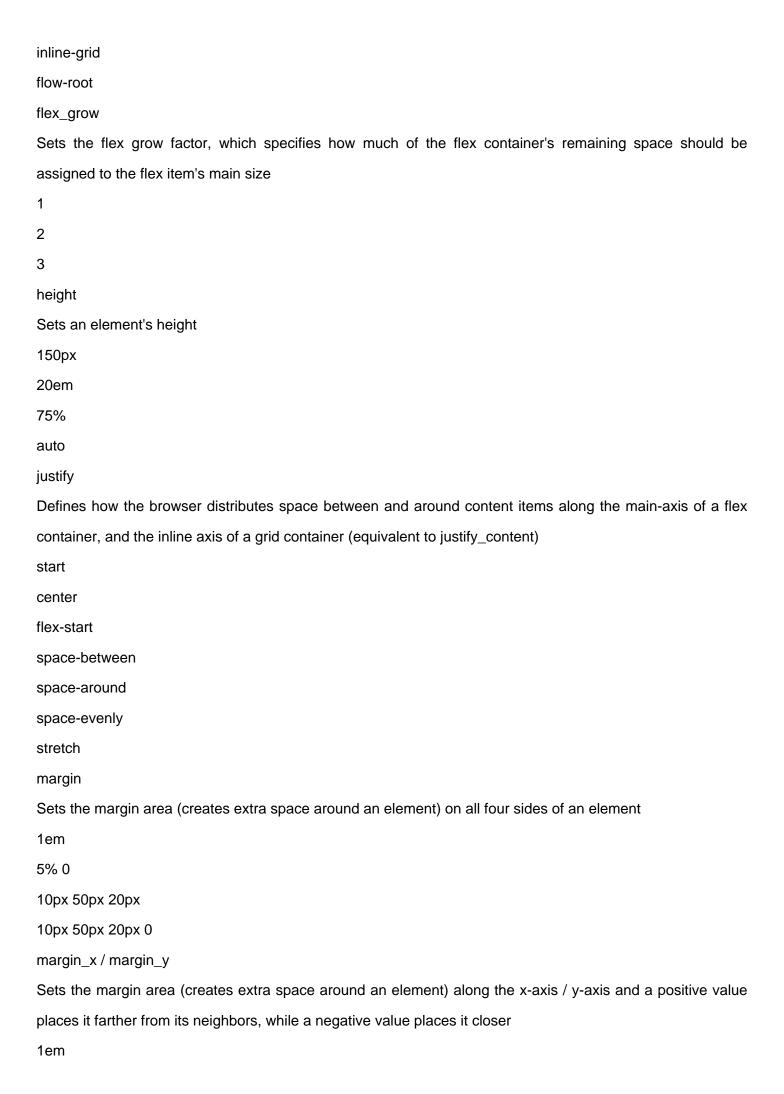
```
Sets all background style properties at once, such as color, image, origin and size, or repeat method
(equivalent to bg)
green
radial-gradient(crimson, skyblue)
no-repeat url('../lizard.png')
background_color
Sets the background color of an element
brown
rgb(255, 255, 128)
#7499ee
background_image
Sets one or more background images on an element
url('../lizard.png')
linear-gradient(#e66465, #9198e5)
border
Sets an element's border, which sets the values of border width, border style, and border color.
solid
dashed red
thick double #32a1ce
4mm ridge rgba(211, 220, 50, .6)
border_top / border_bottom / border_right / border_left
Sets an element's top / bottom / right / left border. It sets the values of border-(top / bottom / right / left)-width,
border-(top / bottom / right / left)-style and border-(top / bottom / right / left)-color
solid
dashed red
thick double #32a1ce
4mm ridge rgba(211, 220, 50, .6)
border color
Sets the color of an element's border (each side can be set individually using border_top_color,
border_right_color, border_bottom_color, and border_left_color)
red
red #32a1ce
red rgba(170, 50, 220, .6) green
red yellow green transparent
```

border\_radius Rounds the corners of an element's outer border edge and you can set a single radius to make circular corners, or two radii to make elliptical corners 30px 25% 10% 10% 30% 50% 70% 10% / 50% border\_width Sets the width of an element's border thick 1em 4px 1.25em 0 4px 8px 12px box\_shadow Adds shadow effects around an element's frame. You can set multiple effects separated by commas. A box shadow is described by X and Y offsets relative to the element, blur and spread radius, and color 10px 5px 5px red 60px -16px teal 12px 12px 2px 1px rgba(0, 0, 255, .2) 3px 3px red, -1em 0 .4em olive; color Sets the foreground color value of an element's text rebeccapurple rgb(255, 255, 128) #00a400 display Sets whether an element is treated as a block or inline box and the layout used for its children, such as flow layout, grid or flex block inline inline-block

flex

grid

inline-flex

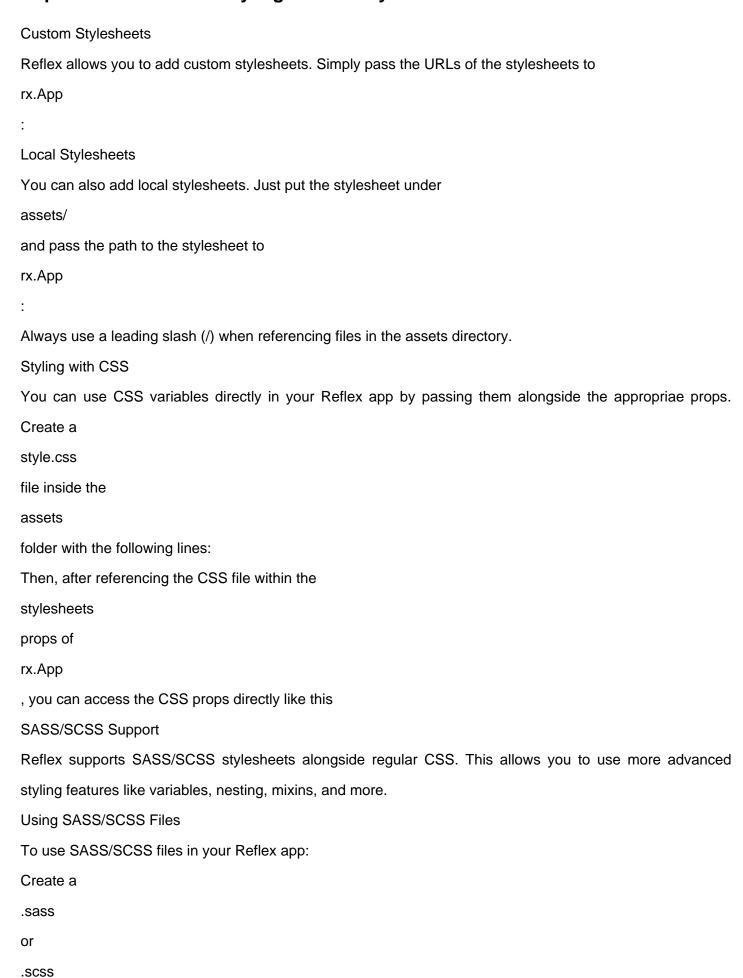


```
10%
10px
margin_top / margin_right / margin_bottom / margin_left
Sets the margin area (creates extra space around an element) on the top / right / bottom / left of an element
1em
10%
10px
max_height / min_height
Sets the maximum / minimum height of an element and prevents the used value of the height property from
becoming larger / smaller than the value specified for max_height / min_height
150px
7em
75%
max_width / min_width
Sets the maximum / minimum width of an element and prevents the used value of the width property from
becoming larger / smaller than the value specified for max width / min width
150px
20em
75%
padding
Sets the padding area (creates extra space within an element) on all four sides of an element at once
1em
10px 50px 30px 0
0
10px 50px 20px
padding_x / padding_y
Creates extra space within an element along the x-axis / y-axis
1em
10%
10px
padding_top / padding_right / padding_bottom / padding_left
Sets the height of the padding area on the top / right / bottom / left of an element
1em
10%
```

20px
position
Sets how an element is positioned in a document and the top, right, bottom, and left properties determine the
final location of positioned elements
static
relative
absolute
fixed
sticky
text_align
Sets the horizontal alignment of the inline-level content inside a block element or table-cell box
start
end
center
justify
left
right
text_wrap
Controls how text inside an element is wrapped
wrap
nowrap
balance
pretty
top / bottom / right / left
Sets the vertical / horizontal position of a positioned element. It does not effect non-positioned elements.
0
4em
10%
20px
width
Sets an element's width
150px
20em
75%

auto
white_space
Sets how white space inside an element is handled
normal
nowrap
pre
break-spaces
word_break
Sets whether line breaks appear wherever the text would otherwise overflow its content box
normal
break-all
keep-all
break-word
z_index
Sets the z-order of a positioned element and its descendants or flex and grid items, and overlapping
elements with a larger z-index cover those with a smaller one
auto
1
5
200

# https://reflex.dev/docs/styling/custom-stylesheets



file in your
assets
directory
Reference the file in your
rx.App
configuration just like you would with CSS files
Reflex automatically detects the file extension and compiles these files to CSS using the
libsass
package.
Example SASS/SCSS File
Here's an example of a SASS file (
assets/styles.scss
) that demonstrates some of the features:
Dependency Requirement
The
libsass
package is required for SASS/SCSS compilation. If it's not installed, Reflex will show an error message. You
can install it with:
This package is included in the default Reflex installation, so you typically don't need to install it separately.
Fonts
You can take advantage of Reflex's support for custom stylesheets to add custom fonts to your app.
In this example, we will use the
IBM Plex Mono
font from Google Fonts. First, add the stylesheet with the font to your app. You can get this link from the "Get
embed code" section of the Google font page.
Then you can use the font in your component by setting the
font_family
prop.
Check out my font
Local Fonts
By making use of the two previous points, we can also make a stylesheet that allow you to use a font hosted
on your server.
If your font is called
MyFont.otf

, copy it in assets/fonts

.

Now we have the font ready, let's create the stylesheet myfont.css

.

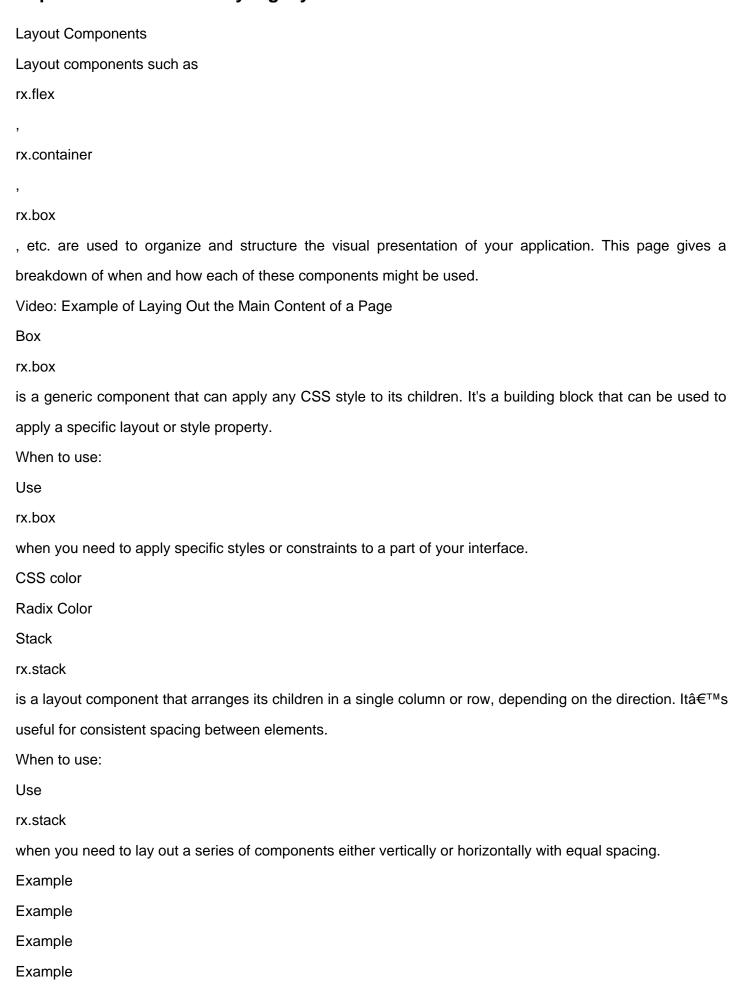
Add the reference to your new Stylesheet in your App.

And that's it! You can now use

MyFont

like any other FontFamily to style your components.

# https://reflex.dev/docs/styling/layout



Flex
The
rx.flex
component is used to create a flexible box layout, inspired by
CSS Flexbox
. It's ideal for designing a layout where the size of the items can grow and shrink dynamically based on the
available space.
When to use:
Use
rx.flex
when you need a responsive layout that adjusts the size and position of child components dynamically.
Card 1
Card 2
Card 3
Grid
rx.grid
components are used to create complex responsive layouts based on a grid system, similar to
CSS Grid Layout
When to use:
Use
rx.grid
when dealing with complex layouts that require rows and columns, especially when alignment and spacing
among multiple axes are needed.
Card 1
Card 2
Card 3
Card 4
Card 5
Card 6
Card 7
Card 8
Card 9
Card 10

Container
The
rx.container
component typically provides padding and fixes the maximum width of the content inside it, often used to
center content on large screens.
When to use:
Use
rx.container
for wrapping your application's content in a centered block with some padding.
This content is constrained to a max width of 448px.

Card 11

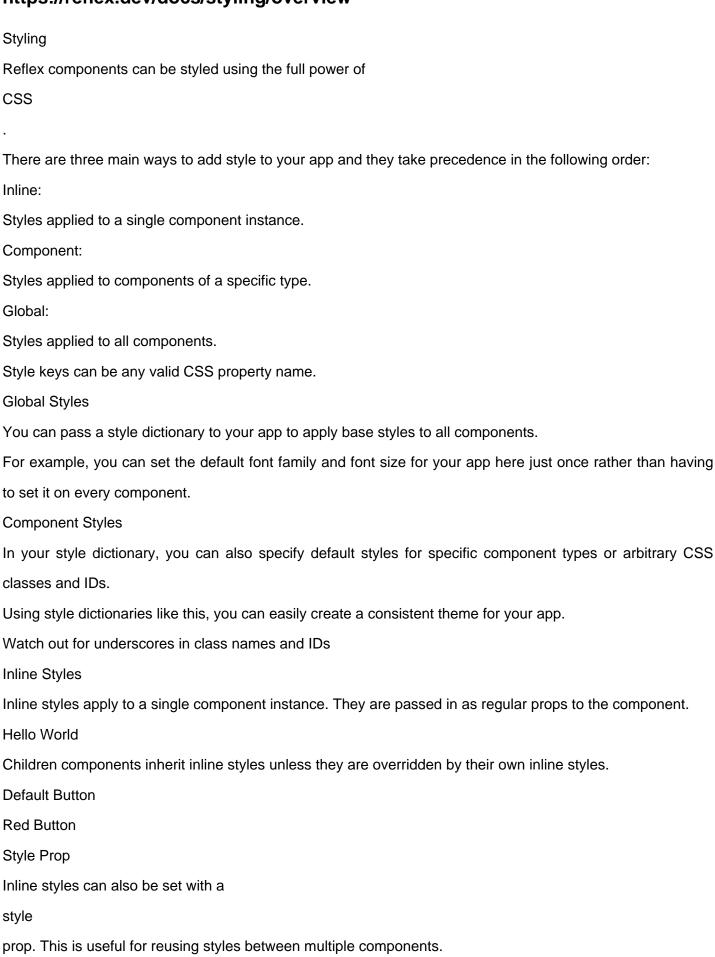
Card 12

This content is constrained to a max width of 688px.

This content is constrained to a max width of 880px.

#### https://reflex.dev/docs/styling/overview

Hello



World Multiple Styles The style dictionaries are applied in the order they are passed in. This means that styles defined later will override styles defined earlier. Theming As of Reflex 'v0.4.0', you can now theme your Reflex web apps. To learn more checkout the Theme docs The Theme component is used to change the theme of the application. The Theme can be set directly in your rx.App. Additionally you can modify the theme of your app through using the Theme Panel component which can be found in the Theme Panel docs Special Styles We support all of Chakra UI's pseudo styles

Below is an example of text that changes color when you hover over it.

Hover Me

# https://reflex.dev/docs/styling/responsive

lg

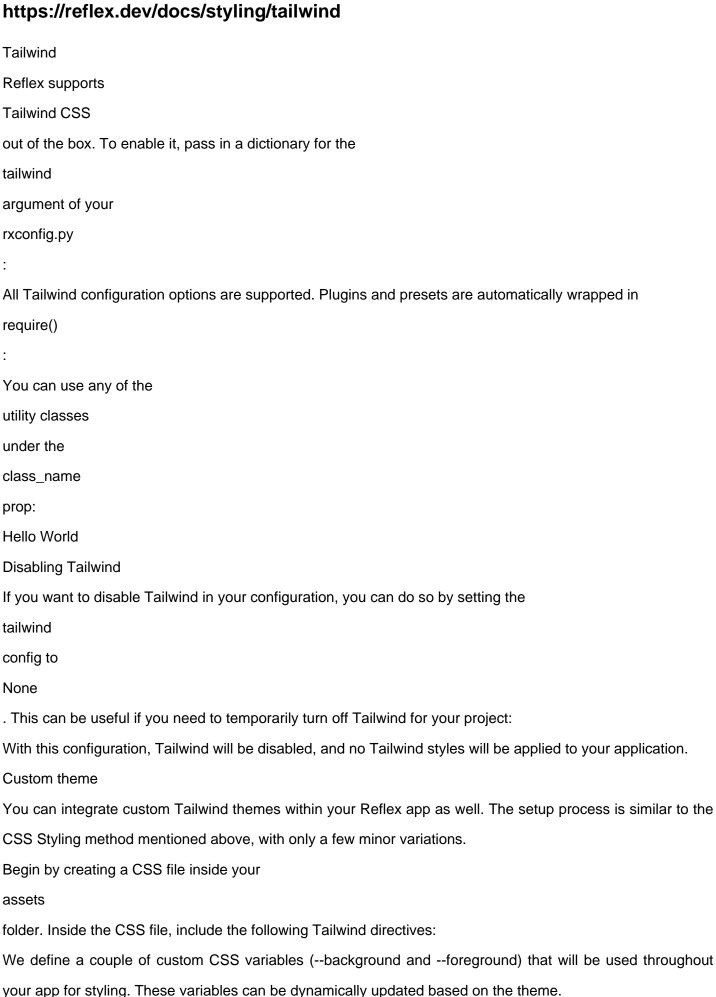
χl

80em

Responsive Reflex apps can be made responsive to look good on mobile, tablet, and desktop. You can pass a list of values to any style property to specify its value on different screen sizes. Hello World The text will change color based on your screen size. If you are on desktop, try changing the size of your browser window to see the color change. New in 0.5.6 Responsive values can also be specified using rx.breakpoints . Each size maps to a corresponding key, the value of which will be applied when the screen size is greater than or equal to the named breakpoint. Hello World Custom breakpoints in CSS units can be mapped to values per component using a dictionary instead of named parameters. For the Radix UI components' fields that supports responsive value, you can also use rx.breakpoints (note that custom breakpoints and list syntax aren't supported). **Setting Defaults** The default breakpoints are shown below. Size Width initial 0px XS 30em sm 48em md 62em

You can customize them using the style property. Showing Components Based on Display A common use case for responsive is to show different components based on the screen size. Reflex provides useful helper components for this. **Desktop View Tablet View** Mobile View Visible on Mobile and Tablet Visible on Desktop and Tablet Specifying Display Breakpoints You can specify the breakpoints to use for the responsive components by using the display style property. Hello World Hello World Hello World Hello World Hello World

96em



Tailwind defaults to light mode, but to handle dark mode, you can define a separate set of CSS variables under the .dark class.

Tailwind Directives (@tailwind base, @tailwind components, @tailwind utilities): These are essential Tailwind CSS imports that enable the default base styles, components, and utility classes.

Next, you'll need to configure Tailwind in your

rxconfig.py

file to ensure that the Reflex app uses your custom Tailwind setup.

In the theme section, we're extending the default Tailwind theme to include custom colors. Specifically, we're referencing the CSS variables (--background and --foreground) that were defined earlier in your CSS file.

The rx.Config object is used to initialize and configure your Reflex app. Here, we're passing the tailwind\_config dictionary to ensure Tailwind's custom setup is applied to the app.

Finally, to apply your custom styles and Tailwind configuration, you need to reference the CSS file you created in your

assets

folder inside the

rx.App

setup. This will allow you to use the custom properties (variables) directly within your Tailwind classes.

In your

app.py

(or main application file), make the following changes:

The bg-background class uses the --background variable (defined in the CSS file), which will be applied as the background color.

Dynamic Styling

You can style a component based of a condition using

rx.cond

or

rx.match

Click me

Using Tailwind Classes from the State

When using Tailwind with Reflex, it's important to understand that class names must be statically defined in your code for Tailwind to properly compile them. If you dynamically generate class names from state variables or functions at runtime, Tailwind won't be able to detect these classes during the build process, resulting in missing styles in your application.

For example, this won't work correctly because the class names are defined in the state:

Click me: false

Using Tailwind with Reflex Core Components

Reflex core components are built on Radix Themes, which means they come with pre-defined styling. When you apply Tailwind classes to these components, you may encounter styling conflicts or unexpected behavior as the Tailwind styles compete with the built-in Radix styles.

For the best experience when using Tailwind CSS in your Reflex application, we recommend using the lower-level

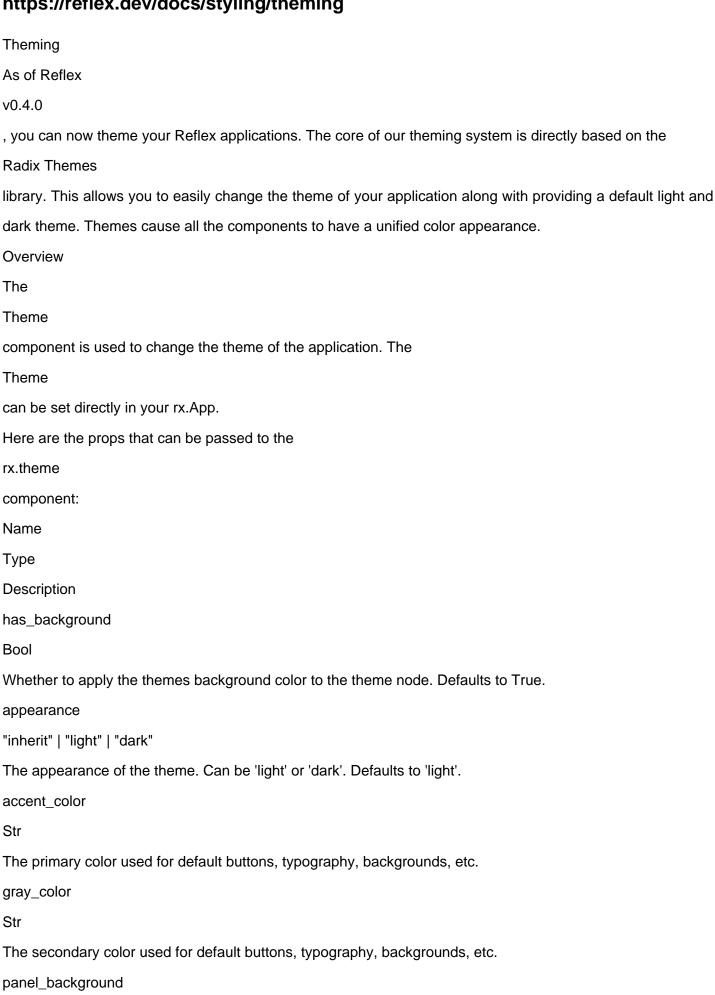
rx.el

components. These components don't have pre-applied styles, giving you complete control over styling with Tailwind classes without any conflicts. Check the list of HTML components

here

.

# https://reflex.dev/docs/styling/theming



```
"solid" | "translucent"
Whether panel backgrounds are translucent: "solid" | "translucent" (default).
radius
"none" | "small" | "medium" | "large" | "full"
The radius of the theme. Can be 'small', 'medium', or 'large'. Defaults to 'medium'.
scaling
"90%" | "95%" | "100%" | "105%" | "110%"
Scale of all theme items.
Additionally you can modify the theme of your app through using the
Theme Panel
component which can be found in the
Theme Panel docs
Colors
Color Scheme
On a high-level, component
color_scheme
inherits from the color specified in the theme. This means that if you change the theme, the color of the
component will also change. Available colors can be found
here
You can also specify the
color scheme
prop.
Hello World
Hello World
Shades
Sometime you may want to use a specific shade of a color from the theme. This is recommended vs using a
hex color directly as it will automatically change when the theme changes appearance change from light/dark.
To access a specific shade of color from the theme, you can use the
```

. When switching to light and dark themes, the color will automatically change. Shades can be accessed by using the color name and the shade number. The shade number ranges from 1 to 12. Additionally, they can have their alpha value set by using the

rx.color

True
parameter it defaults to
False
. A full list of colors can be found
here
Hello World
Name
Туре
Description
color
Str
The color to use. Can be any valid accent color or 'accent' to reference the current theme color.
shade
1 - 12
The shade of the color to use. Defaults to 7.
alpha
Bool
Whether to use the alpha value of the color. Defaults to False.
Regular Colors
You can also use standard hex, rgb, and rgba colors.
Hello World
Toggle Appearance
To toggle between the light and dark mode manually, you can use the
toggle_color_mode
with the desired event trigger of your choice.
Appearance Conditional Rendering
To render a different component depending on whether the app is in
light
mode or
dark
mode, you can use the
rx.color_mode_cond
component. The first component will be rendered if the app is in

light
mode and the second component will be rendered if the app is in
dark
mode.
This can also be applied to props.

Hello World

# https://reflex.dev/docs/ui/overview

#### **UI** Overview

Components are the building blocks for your app's user interface (UI). They are the visual elements that make up your app, like buttons, text, and images.

**Component Basics** 

Components are made up of children and props.

#### Children

- \* Text or other Reflex components nested inside a component.
- \* Passed as \*\*positional arguments\*\*.

#### **Props**

- \* Attributes that affect the behavior and appearance of a component.
- \* Passed as \*\*keyword arguments\*\*.

#### Children

- \* Text or other Reflex components nested inside a component.
- \* Passed as \*\*positional arguments\*\*.

#### **Props**

- \* Attributes that affect the behavior and appearance of a component.
- \* Passed as \*\*keyword arguments\*\*.

Let's take a look at the

rx.text

component.

Hello World!

Here

"Hello World!"

is the child text to display, while

color

and

font size

are props that modify the appearance of the text.

Regular Python data types can be passed in as children to components. This is useful for passing in text, numbers, and other simple data types.

#### Another Example

Now let's take a look at a more complex component, which has other components nested inside it. The

rx.vstack
component is a container that arranges its children vertically with space between them.
Sample Form
Subscribe to Newsletter
Some props are specific to a component. For example, the
header
and
content
props of the
rx.accordion.item
component show the heading and accordion content details of the accordion respectively.
Styling props like
color
are shared across many components.
You can find all the props for a component by checking its documentation page in the
component library
Pages
Reflex apps are organized into pages, each of which maps to a different URL.
Pages are defined as functions that return a component. By default, the function name will be used as the
path, but you can also specify a route explicitly.
In this example we add a page called
index
at the root route.
at the root route.
at the root route.  If you
at the root route.  If you  reflex run
at the root route.  If you reflex run the app, you will see the
at the root route.  If you reflex run the app, you will see the index
at the root route.  If you  reflex run the app, you will see the index page at
at the root route.  If you reflex run the app, you will see the index page at
at the root route.  If you  reflex run  the app, you will see the  index  page at  http://localhost:3000

http://localhost:3000/about

.

## https://reflex.dev/docs/utility-methods/lifespan-tasks

Lifespan Tasks

Added in v0.5.2

Lifespan tasks are coroutines that run when the backend server is running. They are useful for setting up the initial global state of the app, running periodic tasks, and cleaning up resources when the server is shut down.

Lifespan tasks are defined as async coroutines or async contextmanagers. To avoid blocking the event thread, never use

time.sleep

or perform non-async I/O within

a lifespan task.

In dev mode, lifespan tasks will stop and restart when a hot-reload occurs.

**Tasks** 

Any async coroutine can be used as a lifespan task. It will be started when the backend comes up and will run until it returns or is cancelled due to server shutdown. Long-running tasks should catch

asyncio.CancelledError

to perform

any necessary clean up.

Register the Task

To register a lifespan task, use

app.register\_lifespan\_task(coro\_func, \*\*kwargs)

Any keyword arguments specified during registration will be passed to the task.

If the task accepts the special argument,

app

, it will be an instance of the

**FastAPI** 

object

associated with the app.

**Context Managers** 

Lifespan tasks can also be defined as async contextmanagers. This is useful for setting up and tearing down resources and behaves similarly to the ASGI lifespan protocol.

Code up to the first

yield

will run when the backend comes up. As the backend

is shutting down, the code after the

yield

will run to clean up.

Here is an example borrowed from the FastAPI docs and modified to work with this interface.

# https://reflex.dev/docs/utility-methods/other-methods

Other Methods

reset

: set all Vars to their default value for the given state (including substates).

get\_value

: returns the value of a Var

without tracking changes to it

. This is useful

for serialization where the tracking wrapper is considered unserializable.

dict

: returns all state Vars (and substates) as a dictionary. This is used internally when a page is first loaded and needs to be "hydrated" and sent to the client.

**Special Attributes** 

dirty\_vars

: a set of all Var names that have been modified since the last time the state was sent to the client. This is used internally to determine which Vars need to be sent to the client after processing an event.

# https://reflex.dev/docs/utility-methods/router-attributes

State Utility Methods The state object has several methods and attributes that return information about the current page, session, or state. **Router Attributes** The self.router attribute has several sub-attributes that provide various information: router.page : data about the current page and route host : The hostname and port serving the current page (frontend). path : The path of the current page (for dynamic pages, this will contain the slug) raw\_path : The path of the page displayed in the browser (including params and dynamic values) full\_path path with host prefixed full\_raw\_path raw\_path with host prefixed params : Dictionary of query params associated with the request router.session : data about the current session client\_token

: UUID associated with the current tab's token. Each tab has a unique token. session id : The ID associated with the client's websocket connection. Each tab has a unique session ID. client ip : The IP address of the client. Many users may share the same IP address. router.headers : headers associated with the websocket connection. These values can only change when the websocket is re-established (for example, during page refresh). host : The hostname and port serving the websocket (backend). origin : The origin of the request. upgrade : The upgrade header for websocket connections. connection : The connection header. cookie : The cookie header. pragma : The pragma header. cache\_control : The cache control header. user\_agent : The user agent string of the client. sec\_websocket\_version : The websocket version. sec\_websocket\_key : The websocket key. sec\_websocket\_extensions : The websocket extensions. accept\_encoding : The accepted encodings. accept\_language : The accepted languages.

raw headers : A mapping of all HTTP headers as a frozen dictionary. This provides access to any header that was sent with the request, not just the common ones listed above. Example Values on this Page Name Value rx.State.router.page.host rx.State.router.page.path rx.State.router.page.raw\_path rx.State.router.page.full\_path rx.State.router.page.full\_raw\_path rx.State.router.page.params {} rx.State.router.session.client\_token rx.State.router.session.session\_id rx.State.router.session.client\_ip rx.State.router.headers.host rx.State.router.headers.origin rx.State.router.headers.upgrade rx.State.router.headers.connection rx.State.router.headers.cookie rx.State.router.headers.pragma rx.State.router.headers.cache\_control rx.State.router.headers.user\_agent rx.State.router.headers.sec\_websocket\_version rx.State.router.headers.sec\_websocket\_key rx.State.router.headers.sec websocket extensions rx.State.router.headers.accept\_encoding rx.State.router.headers.accept\_language rx.State.router.headers.raw\_headers

Accessing Raw Headers
The

raw\_headers

{}

attribute provides access to all HTTP headers as a frozen dictionary. This is useful when you need to access headers that are not explicitly defined in the

HeaderData

class:

This is particularly useful for accessing custom headers or when working with specific HTTP headers that are not part of the standard set exposed as direct attributes.

# https://reflex.dev/docs/vars/base-vars

. Data associated with a

Base Vars Vars are any fields in your app that may change over time. A Var is directly rendered into the frontend of the app. Base vars are defined as fields in your State class. They can have a preset default value. If you don't provide a default value, you must provide a type annotation. State Vars should provide type annotations. AAPL Current Price: \$150 Change: 4% In this example ticker and price are base vars in the app, which can be modified at runtime. Vars must be JSON serializable. Accessing state variables on different pages State is just a python class and so can be defined on one page and then imported and used on another. Below we define **TickerState** class on the page state.py and then import it and use it on the page index.py **Backend-only Vars** Any Var in a state class that starts with an underscore ( ) is considered backend only and will not be synchronized with the frontend

specific session that is not directly rendered on the frontend should be stored in a backend-only var to reduce network traffic and improve performance. They have the advantage that they don't need to be JSON serializable, however they must still be pickle-able to be used with redis in prod mode. They are not directly renderable on the frontend, and may be used to store sensitive values that should not be sent to the client Protect auth data and sensitive state in backend-only vars. For example, a backend-only var is used to store a large data structure which is then paged to the frontend using cached vars. Prev Page 1 / 10 Next Page Size Generate More  $_{\rm backend}[0] = 80$  $_backend[1] = 69$  $_backend[2] = 56$  $_{backend[3] = 9}$  $_{backend[4] = 13}$  $_{backend[5] = 34}$ \_backend[6] = 57 \_backend[7] = 15 backend[8] = 47 $_{backend[9] = 22}$ Using rx.field / rx.Field to improve type hinting for vars When defining state variables you can use rx.Field[T] to annotate the variable's type. Then, you can initialize the variable using

rx.field(default\_value)

, where

```
default_value
is an instance of type
Т
This approach makes the variable's type explicit, aiding static analysis tools in type checking. In addition, it
shows you what methods are allowed to modify the variable in your frontend code, as they are listed in the
type hint.
Below are two examples:
Here
State.x
, as it is typed correctly as a
boolean
var, gets better code completion, i.e. here we get options such as
to_string()
or
equals()
Here
State.x
, as it is typed correctly as a
dict
of
str
to
list
of
int
var, gets better code completion, i.e. here we get options such as
contains()
keys()
values()
```

items() or

merge()

.

# https://reflex.dev/docs/vars/computed-vars

**Computed Vars** 

Computed vars have values derived from other properties on the backend. They are defined as methods in your State class with the

@rx.var

decorator. A computed

var is recomputed every time an event is processed in your app.

Try typing in the input box and clicking out.

**HELLO** 

Here,

upper\_text

is a computed var that always holds the upper case version of

text

.

We recommend always using type annotations for computed vars.

Cached Vars

A cached var, decorated as

@rx.var(cache=True)

is a special type of computed var

that is only recomputed when the other state vars it depends on change. This is useful for expensive computations, but in some cases it may not update when you expect it to.

Previous versions of Reflex had a

@rx.cached\_var

decorator, which is now replaced

by the new

cache

argument of

@rx.var

.

State touched at: 00:05:14

Counter A: 0 at 00:05:14

Counter B: 0 at 00:05:14

Increment A
Increment B
In this example
last_touch_time
is a normal computed var, which updates any
time the state is modified.
last_counter_a_update
is a computed var that only
depends on
counter_a
, so it only gets recomputed when
counter_a
has changes.
Similarly
last_counter_b_update
only depends on
counter_b
, and thus is
updated only when
counter_b
changes.
Async Computed Vars
Async computed vars allow you to use asynchronous operations in your computed vars.
They are defined as async methods in your State class with the same
@rx.var
decorator.
Async computed vars are useful for operations that require asynchronous processing, such as:
Fetching data from external APIs
Database operations
File I/O operations
Any other operations that benefit from async/await
Async Computed Var Example
Count: 0
Delayed count (x2): 0

Increment

In this example,

delayed\_count

is an async computed var that returns the count multiplied by 2 after a simulated delay.

When the count changes, the async computed var is automatically recomputed.

Caching Async Computed Vars

Just like regular computed vars, async computed vars can also be cached. This is especially useful for expensive async operations like API calls or database queries.

Cached Async Computed Var Example

User ID: 1

User Name: Alice

User Email: alice@example.com

Change User

Force Refresh (No Effect)

Note: The cached async var only updates when user\_id changes, not when refresh\_trigger changes.

In this example,

user\_data

is a cached async computed var that simulates fetching user data.

It is only recomputed when

user\_id

changes, not when other state variables like

refresh\_trigger

change.

This demonstrates how caching works with async computed vars to optimize performance for expensive operations.

# https://reflex.dev/docs/vars/custom-vars

**Custom Vars** 

As mentioned in the

vars page

, Reflex vars must be JSON serializable.

This means we can support any Python primitive types, as well as lists, dicts, and tuples. However, you can also create more complex var types using dataclasses (recommended), TypedDict, or Pydantic models.

Defining a Type

In this example, we will create a custom var type for storing translations using a dataclass.

Once defined, we can use it as a state var, and reference it from within a component.

**Translate** 

Alternative Approaches

Using TypedDict

You can also use TypedDict for defining custom var types:

Using Pydantic Models

Pydantic models are another option for complex data structures:

For complex data structures, dataclasses are recommended as they provide a clean, type-safe way to define custom var types with good IDE support.

#### https://reflex.dev/docs/vars/var-operations

Var Operations Var operations transform the placeholder representation of the value on the frontend and provide a way to perform basic operations on the Var without having to define a computed var. Within your frontend components, you cannot use arbitrary Python functions on the state vars. For example, the following code will not work. This is because we compile the frontend to Javascript, but the value of State.number is only known at runtime. In this example below we use a var operation to concatenate a string with a var , meaning we do not have to do in within state as a computed var. I just bought a bunch of DOGE DOGE is going to the moon! Vars support many common operations. **Supported Operations** Var operations allow us to change vars on the front-end without having to create more computed vars on the back-end in the state. Some simple examples are the var operator, which is used to check if two vars are equal and the to\_string() var operator, which is used to convert a var to a string. "Banana"is my favorite fruit! The selected fruit is not equal to the favorite fruit. Negate, Absolute and Length

operator is used to get the negative version of the var. The

The

abs()
operator is used to get the absolute value of the var. The
.length()
operator is used to get the length of a list var.
The number: 0
Negated:
0
Absolute:
0
Numbers seen:
0
Update
Comparisons and Mathematical Operators
All of the comparison operators are used as expected in python. These include
==
,
!=
,
>
,
>=
,
<
,
<=
•
There are operators to add two vars
+
, subtract two vars
-
, multiply two vars
*
and raise a var to a power
pow()

Integer 1

Integer 2

Operation

Outcome

0

0

Int 1 == Int 2

true

0

0

Int 1 != Int 2

false

0

0

Int 1 > Int 2

false

0

0

Int 1 >= Int 2

true

0

0

Int 1 < Int 2

false

0

0

Int 1 <= Int 2

true

0

0

Int 1 + Int 2

0

0

```
0
Int 1 - Int 2
0
0
Int 1 * Int 2
0
0
0
pow(Int 1, Int2)
1
Update
True Division, Floor Division and Remainder
The operator
represents true division. The operator
//
represents floor division. The operator
%
represents the remainder of the division.
Integer 1
Integer 2
Operation
Outcome
3.5
1.4
Int 1 / Int 2
2.5
3.5
1.4
Int 1 // Int 2
2
3.5
1.4
```

Int 1 % Int 2
0.7000000000002
Update
And, Or and Not
In Reflex the
&
operator represents the logical AND when used in the front end. This means that it returns true only when
both conditions are true simultaneously.
The
I
operator represents the logical OR when used in the front end. This means that it returns true when either
one or both conditions are true.
The
~
operator is used to invert a var. It is used on a var of type
bool
and is equivalent to the
not
operator.
Var 1
Var 2
Operation
Outcome
true
true
Logical AND (&)
true
true
true
Logical OR ( )
true
true
true
The invert of Var 1 (~)

false
Update
Contains, Reverse and Join
The 'in' operator is not supported for Var types, we must use the
Var.contains()
instead. When we use
contains
, the var must be of type:
dict
,
list
,
tuple
or
str
•
contains
checks if a var contains the object that we pass to it as an argument.
We use the
reverse
operation to reverse a list var. The var must be of type
list
•
Finally we use the
join
operation to join a list var into a string.
List 1: 1,2,3,4,6
List 1 Contains 3: true
List 2: 7,8,9,10
Reverse List 2: 10,9,8,7
List 3: p,y,t,h,o,n
List 3 Joins: python
Lower, Upper, Split
The

lower

operator converts a string var to lowercase. The

upper

operator converts a string var to uppercase. The

split

operator splits a string var into a list.

List 1: PYTHON is FUN

List 1 Lower Case: python is fun

List 2: react is hard

List 2 Upper Case: REACT IS HARD

Split String 2: r,e,a,c,t, ,i,s, ,h,a,r,d

Get Item (Indexing)

Indexing is only supported for strings, lists, tuples, dicts, and dataframes. To index into a state var strict type annotations are required.

In the code above you would expect to index into the first index of the list\_1 state var. In fact the code above throws the error:

Invalid var passed for prop value, expected type <class 'int'>, got value of type typing. Any.

This is because the type of the items inside the list have not been clearly defined in the state. To fix this you change the list\_1 definition to

 $list_1: list[int] = [50, 10, 20]$ 

Using with Foreach

Errors frequently occur when using indexing and

foreach

The code above throws the error

TypeError: Could not foreach over var of type Any. (If you are trying to foreach over a state var, add a type annotation to the var.)

We must change

projects: list[dict]

=>

projects: list[dict[str, list]]

because while projects is annotated, the item in project["technologies"] is not.

Next.js

Prisma

Tailwind
Google Cloud
Docker
MySQL
Python
Flask
Google Cloud
Docker
The previous example had only a single type for each of the dictionaries
keys
and
values
For complex multi-type data, you need to use a dataclass, as shown below.
Ariana Grande
30
arianagrande.com
nttps://es.wikipedia.org/wiki/Ariana_Grande
Gal Gadot
38
http://www.galgadot.com/
nttps://es.wikipedia.org/wiki/Gal_Gadot
Setting the type of
actresses
to be
actresses: list[dict[str,str]]
would fail as it cannot be understood that the
value
for the
pages key
s actually a
ist

Combine Multiple Var Operations

You can also combine multiple var operations together, as seen in the next example.

The number is 0

Update

Even

We could have made a computed var that returns the parity of number

, but

it can be simpler just to use a var operation instead.

Var operations may be generally chained to make compound expressions, however some complex transformations not supported by var operations must use computed vars to calculate the value on the backend.

# https://reflex.dev/docs/wrapping-react/custom-code-and-hooks

When wrapping a React component, you may need to define custom code or hooks that are specific to the
component. This is done by defining the
add_custom_code
or
add_hooks
methods in your component class.
Custom Code
Custom code is any JS code that need to be included in your page, but not necessarily in the componen
itself. This can include things like CSS styles, JS libraries, or any other code that needs to be included in the
page.
The above example will render the following JS code in the page:
Custom Hooks
Custom hooks are any hooks that need to be included in your component. This can include things like
useEffect
,
useState
, or any other hooks from the library you are wrapping.
Simple hooks can be added as strings.
More complex hooks that need to have special import or be written in a specific order can be added as
rx.Var
with a
VarData
object to specify the position of the hook.
The
imports
attribute of the
VarData
object can be used to specify any imports that need to be included in the component.
The
position
attribute of the
VarData

object can be set to

Hooks.HookPosition.PRE\_TRIGGER

or

Hooks.HookPosition.POST\_TRIGGER

to specify the position of the hook in the component.

The

position

attribute is only used for hooks that need to be written in a specific order.

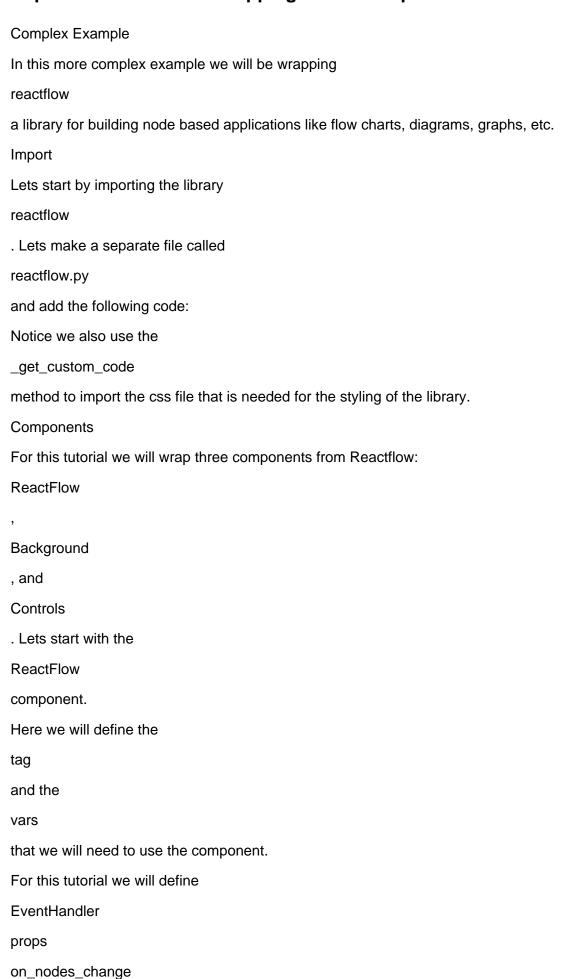
The

ComponentWithHooks

will be rendered in the component in the following way:

You can mix custom code and hooks in the same component. Hooks can access a variable defined in the custom code, but custom code cannot access a variable defined in a hook.

# https://reflex.dev/docs/wrapping-react/example



```
and
on_connect
, but you can find all the events that the component triggers in the
reactflow docs
Now lets add the
Background
and
Controls
components. We will also create the components using the
create
method so that we can use them in our app.
Building the App
Now that we have our components lets build the app.
Lets start by defining the initial nodes and edges that we will use in our app.
Next we will define the state of our app. We have four event handlers:
add_random_node
clear_graph
on_connect
and
on_nodes_change
The
on_nodes_change
event handler is triggered when a node is selected and dragged. This function is used to update the position
of a node during dragging. It takes a single argument
node_changes
, which is a list of dictionaries containing various types of metadata. For updating positions, the function
specifically processes changes of type
position
Now lets define the UI of our app. We will use the
```

react_flow
component and pass in the
nodes
and
edges
from our state. We will also add the
on_connect
event handler to the
react_flow
component to handle when an edge is connected.
Here is an example of the app running:
React Flow
Press enter or space to select a node.
You can then use the arrow keys to move the node around.
Press delete to remove it and escape to cancel.
Press enter or space to select an edge. You can then press delete to remove it or escape to cancel.
Clear graph
Add node

## https://reflex.dev/docs/wrapping-react/imports-and-styles

Styles and Imports

When wrapping a React component, you may need to define styles and imports that are specific to the component. This is done by defining the

add\_styles

and

add\_imports

methods in your component class.

**Imports** 

Sometimes, the component you are wrapping will need to import other components or libraries. This is done by defining the

add\_imports

method in your component class.

That method should return a dictionary of imports, where the keys are the names of the packages to import and the values are the names of the components or libraries to import.

Values can be either a string or a list of strings. If the import needs to be aliased, you can use the ImportVar

object to specify the alias and whether the import should be installed as a dependency.

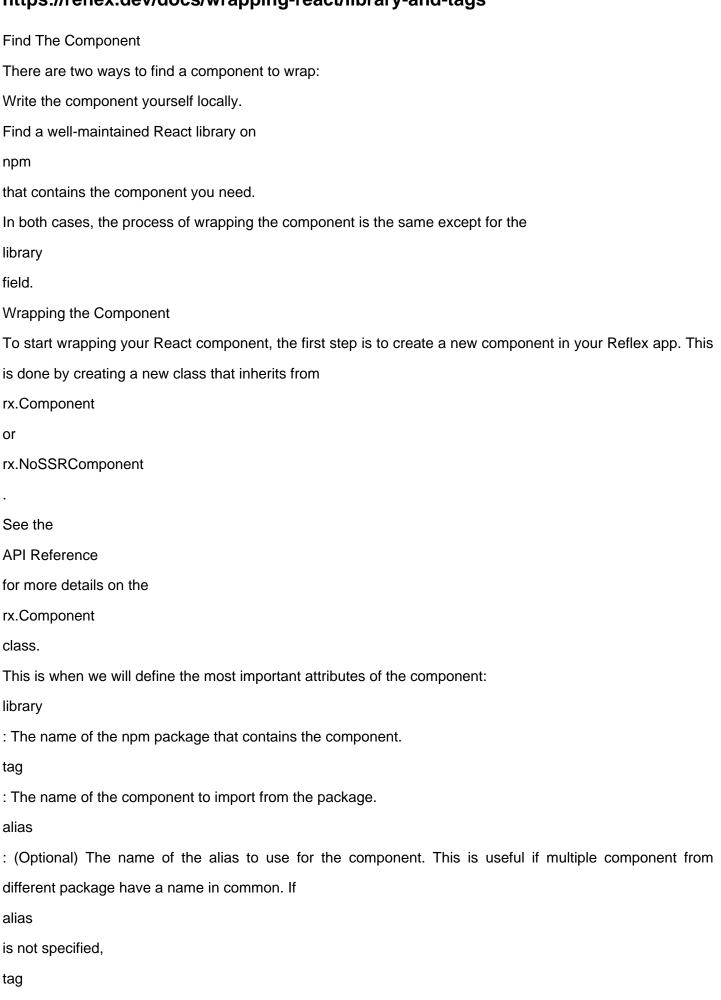
The tag and library of the component will be automatically added to the imports. They do not need to be added again in

add\_imports

Styles

Styles are any CSS styles that need to be included in the component. The style will be added inline to the component, so you can use any CSS styles that are valid in React.

# https://reflex.dev/docs/wrapping-react/library-and-tags



will be used. lib dependencies : Any additional libraries needed to use the component. is default : (Optional) If the component is a default export from the module, set this to True . Default is False Optionally, you can override the default component creation behavior by implementing the create class method. Most components won't need this when props are straightforward conversions from Python to JavaScript. However, this is useful when you need to add custom initialization logic, transform props, or handle special cases when the component is created. When setting the library attribute, it is recommended to included a pinned version of the package. Doing so, the package will only change when you intentionally update the version, avoid unexpected breaking changes. Wrapping a Dynamic Component When wrapping some libraries, you may want to use dynamic imports. This is because they may not be compatible with Server-Side Rendering (SSR). To handle this in Reflex, subclass NoSSRComponent when defining your component. It works the same as rx.Component , but it will automatically add the correct custom code for a dynamic import. Often times when you see an import something like this: You can wrap it in Reflex like this: It may not always be clear when a library requires dynamic imports. A few things to keep in mind are if the component is very client side heavy i.e. the view and structure depends on things that are fetched at run time,

or if it uses

document

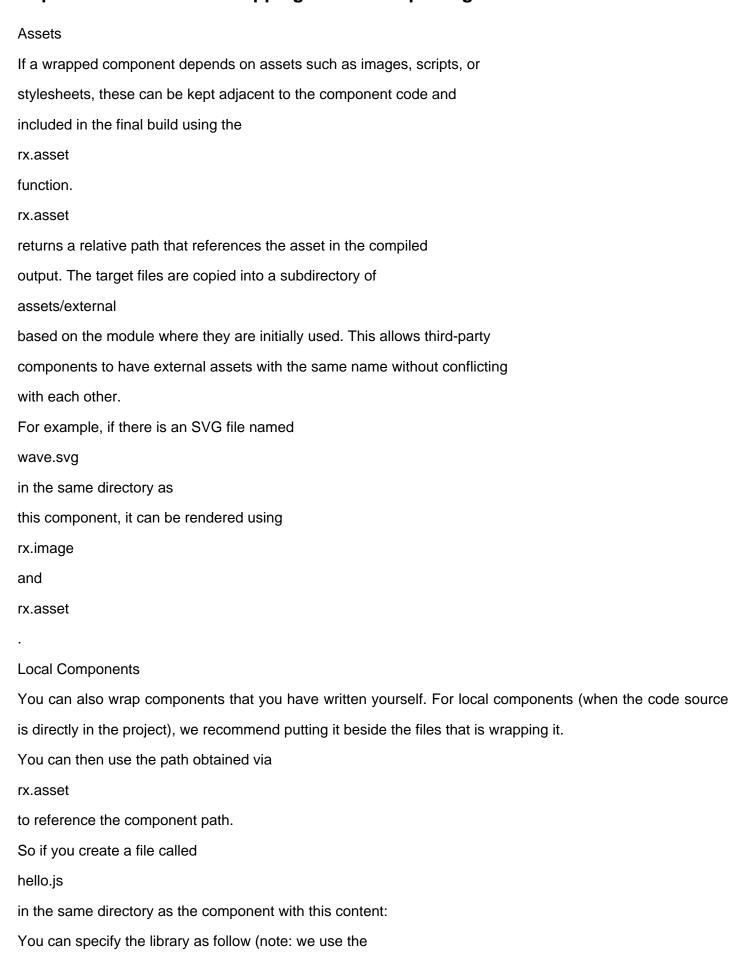
window

or

objects directly it will need to be wrapped as a
NoSSRComponent
Some examples are:
Video and Audio Players
Maps
Drawing Canvas
3D Graphics
QR Scanners
Reactflow
The reason for this is that it does not make sense for your server to render these components as the server
does not have access to your camera, it cannot draw on your canvas or render a video from a file.
In addition, if in the component documentation it mentions nextJS compatibility or server side rendering
compatibility, it is a good sign that it requires dynamic imports.
Advanced - Parsing a state Var with a JS Function
When wrapping a component, you may need to parse a state var by applying a JS function to it.
Define the parsing function
First you need to define the parsing function by writing it in
add_custom_code
•
Apply the parsing function to your props
Then, you can apply the parsing function to your props in the
create
method.

## https://reflex.dev/docs/wrapping-react/local-packages

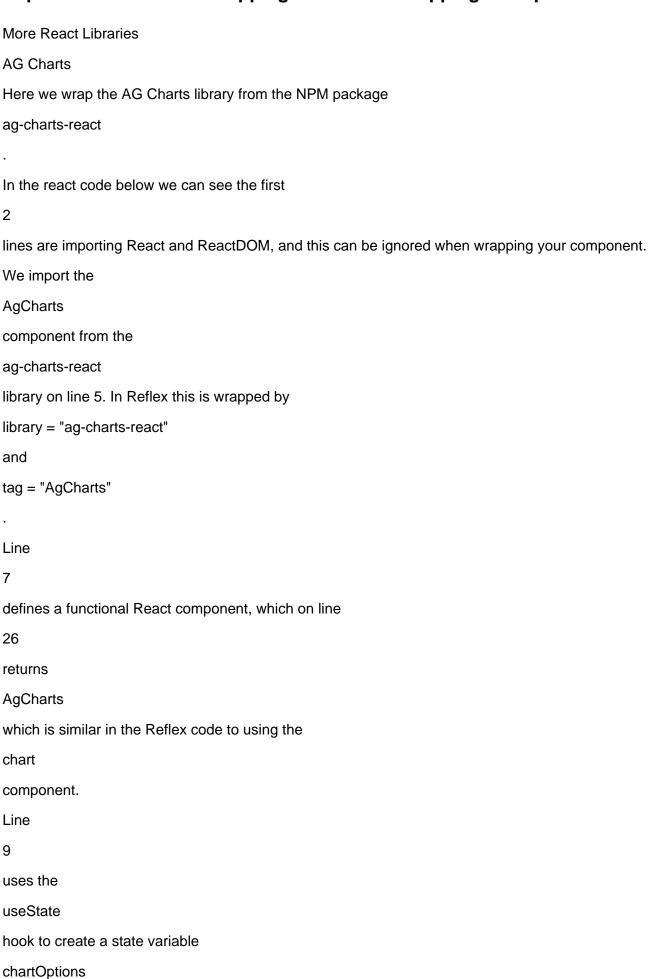
public



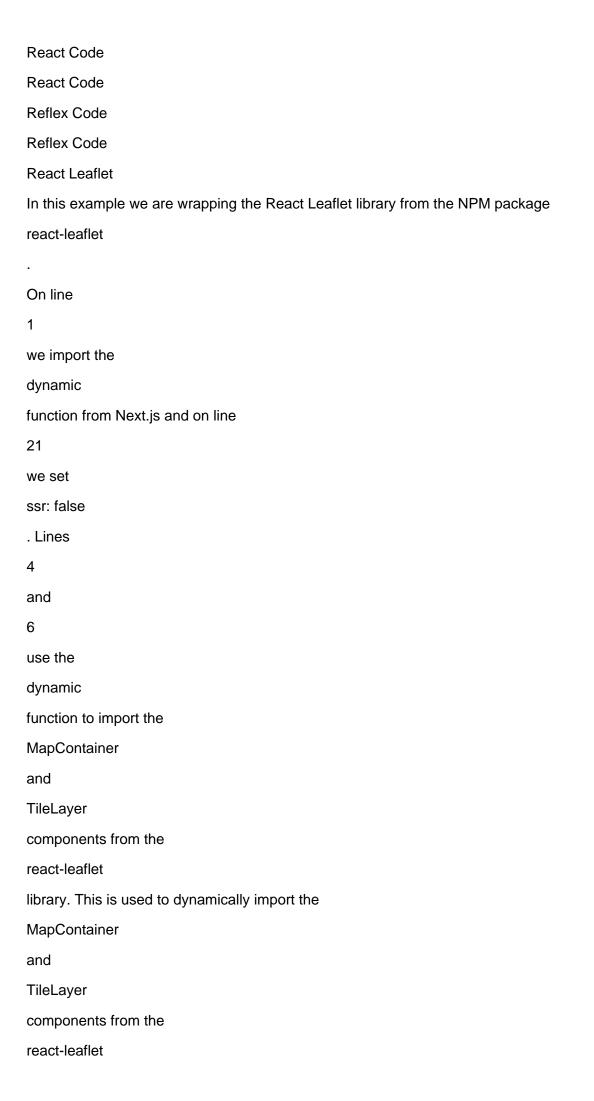
directory here instead of assets as this is the directory that is served by the web server): **Local Packages** If the component is part of a local package, available on Github, or downloadable via a web URL, it can also be wrapped in Reflex. Specify the path or URL after an @ following the package name. Any local paths are relative to the .web folder, so you can use ../ prefix to reference the Reflex project root. Some examples of valid specifiers for a package called @masenf/hello-react are: GitHub: @masenf/hello-react@github:masenf/hello-react **URL**: @masenf/hello-react@https://github.com/masenf/hello-react/archive/refs/heads/main.tar.gz Local Archive: @masenf/hello-react@../hello-react.tgz **Local Directory:** @masenf/hello-react@../hello-react It is important that the package name matches the name in package.json so Reflex can generate the correct import statement in the generated javascript code. These package specifiers can be used for library or

lib_dependencies
UI
UI
Code
Code
Counter
Increment by one
Total value:
0
Although more complicated, this approach is useful when the local components
have additional dependencies or build steps required to prepare the component
for use.
Some important notes regarding this approach:
The repo or archive must contain a
package.json
file.
prepare
or
build
scripts will NOT be executed. The distribution archive,
directory, or repo must already contain the built javascript files (this is common).
Ensure CSS files are exported in
package.json

## https://reflex.dev/docs/wrapping-react/more-wrapping-examples



and its setter function
setChartOptions
(equivalent to the event handler
set_chart_options
in reflex). The initial state variable is of type dict and has two key value pairs
data
and
series
When we see
useState
in React code, it correlates to state variables in your State. As you can see in our Reflex code we have a
state variable
chart_options
which is a dictionary, like in our React code.
Moving to line
26
we see that the
AgCharts
has a prop
options
. In order to use this in Reflex we must wrap this prop. We do this with
options: rx.Var[dict]
in the
AgCharts
component.
Lines
31
and
32
are rendering the component inside the root element. This can be ignored when we are wrapping a
component as it is done in Reflex by creating an
index
function and adding it to the app.



library. This is done in Reflex by using the
NoSSRComponent
class when defining the component. There is more information of when this is needed on the
Dynamic Imports
section of this
page
It mentions in the documentation that it is necessary to include the Leaflet CSS file, which is added on line
2
in the React code below. This can be done in Reflex by using the
add_imports
method in the
MapContainer
component. We can add a relative path from within the React library or a full URL to the CSS file.
Line
4
defines a functional React component, which on line
8
returns the
MapContainer
which is done in the Reflex code using the
map_container
component.
The
MapContainer
component has props
center
,
zoom
,
scrollWheelZoom
, which we wrap in the
MapContainer
component in the Reflex code. We ignore the

style
prop as it is a reserved name in Reflex. We can use the
rename_props
method to change the name of the prop, as we will see in the React PDF Renderer example, but in this case
we just ignore it and add the
width
and
height
props as css in Reflex.
The
TileLayer
component has a prop
url
which we wrap in the
TileLayer
component in the Reflex code.
Lines
24
and
25
defines and exports a React functional component named
Home
which returns the
MapComponent
component. This can be ignored in the Reflex code when wrapping the component as we return the
map_container
component in the
index
function.
React Code
React Code
Reflex Code
Reflex Code

React PDF Renderer

In this example we are wrapping the React renderer for creating PDF files on the browser and server from the
NPM package
@react-pdf/renderer
This example is similar to the previous examples, and again Dynamic Imports are required for this library.
This is done in Reflex by using the
NoSSRComponent
class when defining the component. There is more information on why this is needed on the
Dynamic Imports
section of this
page
The main difference with this example is that the
style
prop, used on lines
20
,
21
and
24
in React code, is a reserved name in Reflex so can not be wrapped. A different name must be used when
wrapping this prop and then this name must be changed back to the original with the
rename_props
method. In this example we name the prop
theme
in our Reflex code and then change it back to
style
with the
rename_props
method in both the
Page
and
View
components.

List of reserved names in Reflex
React Code
React Code

Reflex Code

Reflex Code

### https://reflex.dev/docs/wrapping-react/overview

Wrapping React

One of Reflex's most powerful features is the ability to wrap React components and take advantage of the vast ecosystem of React libraries.

If you want a specific component for your app but Reflex doesn't provide it, there's a good chance it's available as a React component. Search for it on

npm

, and if it's there, you can use it in your Reflex app. You can also create your own local React components and wrap them in Reflex.

Once you wrap your component, you

publish it

to the Reflex library so that others can use it.

Simple Example

Simple components that don't have any interaction can be wrapped with just a few lines of code.

Below we show how to wrap the

Spline

library can be used to create 3D scenes and animations.

ColorPicker Example

Similar to the Spline example we start with defining the library and tag. In this case the library is

react-colorful

and the tag is

HexColorPicker

We also have a var

color

which is the current color of the color picker.

Since this component has interaction we must specify any event triggers that the component takes. The color picker has a single trigger

on\_change

to specify when the color changes. This trigger takes in a single argument

color

which is the new color.

#db114b

What Not To Wrap

There are some libraries on npm that are not do not expose React components and therefore are very hard to wrap with Reflex.

A library like

spline

below is going to be difficult to wrap with Reflex because it does not expose a React component.

You should look out for JSX, a syntax extension to JavaScript, which has angle brackets (<h1>Hello, world!</h1>)

. If you see JSX, it's likely that the library is a React component and can be wrapped with Reflex.

If the library does not expose a react component you need to try and find a JS React wrapper for the library,

react-spline

such as

In the next page, we will go step by step through a more complex example of wrapping a React component.

### https://reflex.dev/docs/wrapping-react/props

Props

When wrapping a React component, you want to define the props that will be accepted by the component.

This is done by defining the props and annotating them with a

rx.Var

.

Broadly, there are three kinds of props you can encounter when wrapping a React component:

Simple Props

: These are props that are passed directly to the component. They can be of any type, including strings, numbers, booleans, and even lists or dictionaries.

Callback Props

: These are props that expect to receive a function. That function will usually be called by the component as a callback. (This is different from event handlers.)

Component Props

: These are props that expect to receive a components themselves. They can be used to create more complex components by composing them together.

**Event Handlers** 

: These are props that expect to receive a function that will be called when an event occurs. They are defined as

rx.EventHandler

with a signature function to define the spec of the event.

Simple Props

Simple props are the most common type of props you will encounter when wrapping a React component.

They are passed directly to the component and can be of any type (but most commonly strings, numbers, booleans, and structures).

For custom types, you can use

**TypedDict** 

to define the structure of the custom types. However, if you need the attributes to be automatically converted to camelCase once compiled in JS, you can use

rx.PropsBase

instead of

TypedDict

.

Callback Props

Callback props are used to handle events or to pass data back to the parent component. They are defined as rx.Var

with a type of

FunctionVar

or

Callable

.

Component Props

Some components will occasionally accept other components as props, usually annotated as

ReactNode

. In Reflex, these are defined as

rx.Component

.

**Event Handlers** 

Event handlers are props that expect to receive a function that will be called when an event occurs. They are defined as

rx.EventHandler

with a signature function to define the spec of the event.

Custom event specs have a few use case where they are particularly useful. If the event returns non-serializable data, you can filter them out so the event can be sent to the backend. You can also use them to transform the data before sending it to the backend.

# https://reflex.dev/docs/wrapping-react/serializers

#### Serializers

Vars can be any type that can be serialized to JSON. This includes primitive types like strings, numbers, and booleans, as well as more complex types like lists, dictionaries, and dataframes.

In case you need to serialize a more complex type, you can use the serializer

decorator to convert the type to a primitive type that can be stored in the state. Just define a method that takes the complex type as an argument and returns a primitive type. We use type annotations to determine the type that you want to serialize.

For example, the Plotly component serializes a plotly figure into a JSON string that can be stored in the state. We can then define a var of this type as a prop in our component.