Unicorn

version

Adam Hill

August 12, 2023

Contents

Introduction

Installation

Install `Unicorn`

Install `Unicorn` the same as any other Python package (preferably into a virtual environment).

```
pip install django-unicorn
```

OR

```
poetry add django-unicorn
```

> ***Note***
>
> If attempting to install `django-unicorn` and `orjson` is preventing the installation from succeeding, check whether it is using 32-bit Python. Unfortunately, `orjson` is only supported on 64-bit Python. More details in issue #105.

Integrate `Unicorn` *with Django*

1. Add `django_unicorn` to the `INSTALLED_APPS` list in the Django settings file (normally `settings.py`).

```
# settings.py
INSTALLED_APPS = (
    # other apps
    "django_unicorn",  # required for Django to register urls and templatetags
    # other apps
)
```

2. Add `path("unicorn/", include("django_unicorn.urls")),` into the project's `urls.py`.

```
# urls.py
urlpatterns = (
    # other urls
    path("unicorn/", include("django_unicorn.urls")),
)
```

3. Add `{% load unicorn %}` to the top of the Django HTML template.

4. Add `{% unicorn_scripts %}` into the Django HTML template and make sure there is a `{% csrf_token %}` in the template as well.

```
<!-- index.html -->
{% load unicorn %}
<html>
  <head>
    {% unicorn_scripts %}
  </head>
  <body>
    {% csrf_token %}
  </body>
</html>
```

Then, create a component.

Components

`Unicorn` uses the term "component" to refer to a set of interactive functionality that can be put into templates. A component consists of a Django HTML `template` with specific tags and a Python `view` class which provides the backend code for the template.

Create a component

The easiest way to create your first component is to run the `startunicorn` Django management command after `Unicorn` is installed.

The first argument to `startunicorn` is the Django app to add your component to. Every argument after is a new component to create a template and view for.

```
# Create `hello-world` and `hello-magic` components in a `unicorn` app
python manage.py startunicorn unicorn hello-world hello-magic
```

Warning

If the app does not already exist, `startunicorn` will ask if it should call `startapp` to create a new application. However, make sure to add the app name to `INSTALLED_APPS` in your Django settings file (normally `settings.py`). Otherwise Django will not be able to find the newly created component templates.

Note

Explicitly set which apps `Unicorn` looks in for components with the APPS setting. Otherwise, all `INSTALLED_APPS` will be searched for components.

Then, add a `{% unicorn 'hello-world' %}` templatetag into the template where you want to load the new component.

Warning

Make sure that there is a `{% csrf_token %}` rendered by the HTML template that includes the component to prevent cross-site scripting attacks while using `Unicorn`.

Component key

If there are multiple of the same components on the page, a `key` kwarg can be passed into the template. For example, `{% unicorn 'hello-world' key='helloWorldKey' %}`. This is useful when a unique reference to a component is required, but it is optional.

Component arguments

`args` and `kwargs` can be passed into the `unicorn` templatetag from the template. They will be available in the component `component_args` and `component_kwargs` methods respectively.

```
<!-- index.html -->
{% unicorn 'hello-world' "Hello" name="World" %}
```

```
# hello_world.py
from django_unicorn.components import UnicornView

class HelloWorldView(UnicornView):
```

```
    def mount(self):
        arg = self.component_args[0]
        kwarg = self.component_kwargs["name"]

        assert f"{arg} {kwarg}" == "Hello World"
```

Regular Django template variables can also be passed in as an argument as long as it is available in the template context.

```
<!-- index.html -->
{% unicorn 'hello-world' name=hello.world.name %}
```

```
# views.py
from django.shortcuts import render

def index(request):
    context = {"hello": {"world": {"name": "Galaxy"}}}
    return render(request, "index.html", context)
```

```
class HelloWorldView(UnicornView):
    def mount(self):
        kwarg = self.component_kwargs["name"]

        assert kwarg == "Galaxy"
```

Example component

A basic example component could consist of the following template and class.

```
# hello_world.py
from django_unicorn.components import UnicornView

class HelloWorldView(UnicornView):
    name = "World"
```

```
<!-- hello-world.html -->
<div>
  <input unicorn:model="name" type="text" id="text" /><br />
  Hello {{ name|title }}
</div>
```

Warning

`Unicorn` requires there to be one root element surrounding the component template.

`unicorn:model` is the magic that ties the input to the backend component. The Django template variable can use any property or method on the component as if they were context variables passed in from a view. The attribute passed into `unicorn:model` refers to the property in the component class and binds them together.

Note

By default `unicorn:model` updates are triggered by listening to `input` events on the element. To listen for the `blur` event instead, use the lazy modifier.

When a user types into the text input, the information is passed to the backend and populates the component class, which is then used to generate the output of the template HTML. The template can use any normal Django templatetags or filters (e.g. the `title` filter above).

Component sub-folders

Components can also be nested in sub-folders.

```
unicorn/
    components/
        __init__.py
        hello/
            __init__.py
            world.py
    templates/
        unicorn/
            hello/
                world.html
```

An example of how the above component would be included in a template.

```
<!-- index.html -->
{% unicorn 'hello.world' %}
```

Unicorn attributes

Attributes used in component templates usually start with `unicorn:`, however the shortcut `u:` is also supported. So, for example, `unicorn:model` could also be written as `u:model`.

Supported property types

Properties of the component can be of many different types, including `str`, `int`, `list`, `dictionary`, `Decimal`,`Django Model`, `Django QuerySet`, `dataclass`, or `custom classes`.

Property type hints

`Unicorn` will attempt to cast any properties with a `type hint` when the component is hydrated.

```python
# rating.py
from django_unicorn.components import UnicornView

class RatingView(UnicornView):
    rating: float = 0

    def calculate_percentage(self):
        print(self.rating / 100.0)
```

Without `rating: float`, when `calculate_percentage` is called Python will complain with an error message like the following.

```
TypeError: unsupported operand type(s) for /: 'str' and 'int'`
```

Accessing nested fields

Fields in a `dictionary` or Django model can be accessed similarly to the Django template language with “dot-notation”.

```python
# hello_world.py
from django_unicorn.components import UnicornView
from book.models import Book

class HelloWorldView(UnicornView):
```

```
    book = Book.objects.get(title='American Gods')
    book_ratings = {'excellent': {'title': 'American Gods'}}
```

```
<!-- hello-world.html -->
<div>
  <input unicorn:model="book.title" type="text" id="model" />
  <input
    unicorn:model="book_ratings.excellent.title"
    type="text"
    id="dictionary"
  />
</div>
```

Note

Django models has many more details about using Django models in `Unicorn`.

Django QuerySet

`Django QuerySet` can be referenced similarly to the Django template language in a `unicorn:model`.

```
# hello_world.py
from django_unicorn.components import UnicornView
from book.models import Book

class HelloWorldView(UnicornView):
    books = Book.objects.all()
```

```
<!-- hello-world.html -->
<div>
  <input unicorn:model="books.0.title" type="text" id="text" />
</div>
```

Note

Django models has many more details about using Django QuerySets in `Unicorn`.

Custom class

Custom classes need to define how they are serialized. If you have access to the object to serialize, you can define a `to_json` method on the object to return a dictionary that can be used to serialize. Inheriting from `unicorn.components.UnicornField` is a quick way to serialize a custom class, but note that it just calls `self.__dict__` under the hood, so it is not doing anything particularly smart.

Another option is to set the `form_class` on the component and utilize Django's built-in forms and widgets to handle how the class should be deserialized. More details are provided in validation.

```
# hello_world.py
from django_unicorn.components import UnicornView, UnicornField

class Author(UnicornField):
    def mount(self):
        self.name = 'Neil Gaiman'

    # Not needed because inherited from `UnicornField`
```

```
    # def to_json(self):
    #     return {'name': self.name}

    class HelloWorldView(UnicornView):
        author = Author()
```

```
<!-- hello-world.html -->
<div>
  <input unicorn:model="author.name" type="text" id="author_name" />
</div>
```

!DANGER!

Never put sensitive data into a public property because that information will publicly available in the HTML source code, unless explicitly prevented with `javascript_exclude`.

Templates

Templates are just normal Django HTML templates, so anything you could normally do in a Django template will still work, including template tags, filters, loops, if statements, etc.

Warning

`Unicorn` requires there to be one root element surrounding the component template.

Note

To reduce the verbosity of templates, `u:` can be used as a shorthand for any attribute that starts with `unicorn:`. All of the examples in the documentation use `unicorn:` to be explicit, but both are supported.

Model modifiers

Lazy

To prevent updates from happening on *every* input, you can append a `lazy` modifier to the end of `unicorn:model`. That will only update the component when a `blur` event happens.

```
<!-- waits-for-blur.html -->
<div>
  <input unicorn:model.lazy="name" type="text" id="name" />
  Hello {{ name|title }}
</div>
```

Debounce

The `debounce` modifier configures how long to wait to fire an event. The time is always specified in milliseconds, for example: `unicorn:model.debounce-1000` would wait for 1000 milliseconds (1 second) before firing the message.

```
<!-- waits-1-second.html -->
<div>
  <input unicorn:model.debounce-1000="name" type="text" id="name" />
```

```
  Hello {{ name|title }}
</div>
```

Defer

The `defer` modifier will store and save model changes until the next action gets triggered. This is useful to prevent additional network requests until an action is triggered.

```
<!-- defer.html -->
<div>
  <input unicorn:model.defer="task" type="text" id="task" />
  <button unicorn:click="add">Add task</button>
  <ul>
    {% for task in tasks %}
    <li>{{ task }}</li>
    {% endfor %}
  </ul>
</div>
```

Chaining modifiers

`Lazy` and `debounce` modifiers can even be chained together.

```
<!-- waits-for-blur-and-then-5-seconds.html -->
<div>
  <input unicorn:model.lazy.debounce-5000="name" type="text" id="text" />
  Hello {{ name|title }}
</div>
```

Key

Smooth updates

Setting a unique `id` on elements with `unicorn:model` will prevent changes to an input from being choppy when there are lots of updates in quick succession.

```
<!-- choppy-updates.html -->
<input type="text" unicorn:model="name"></input>
```

```
!-- smooth-updates.html -->
<input type="text" id="someFancyId" unicorn:model="name"></input>
```

However, setting the same `id` on two elements with the same `unicorn:model` won't work. The `unicorn:key` attribute can be used to make sure that the elements can be synced as expected.

```
<!-- missing-updates.html -->
<input type="text" id="someFancyId" unicorn:model="name"></input>
<input type="text" id="someFancyId" unicorn:model="name"></input>
```

```
<!-- this-should-work.html -->
<input type="text" id="someFancyId" unicorn:model="name"></input>
<input type="text" id="someFancyId" unicorn:model="name" unicorn:key="someFancyKey"></input>
```

DOM merging

The JavaScript library used to merge changes in the DOM, `morphdom`, uses an element's `id` to intelligently update DOM elements. If it isn't possible to have an `id` attribute on the element, `unicorn:key` will be used if it is available.

Lifecycle events

`Unicorn` provides events that fire when different parts of the lifecycle occur.

updated

The `updated` event is fired after the AJAX call finishes and the component is merged with the newly rendered component template. The callback gets called with one argument, `component`, which can be inspected if necessary.

```
<!-- updated-event.html -->

<script type="application/javascript">
  window.addEventListener("DOMContentLoaded", (event) => {
    Unicorn.addEventListener("updated", (component) =>
      console.log("got updated", component)
    );
  });
</script>
```

Ignore elements

Some JavaScript libraries will change the DOM (such as `Select2`) after the page renders. That can cause issues for `Unicorn` when trying to merge that DOM with what `Unicorn` *thinks* the DOM should be. `unicorn:ignore` can be used to prevent `Unicorn` from morphing that element or its children.

Note

When the component is initially rendered, normal Django template functionality can be used.

```
<!-- ignored-element.html -->
<div>
  <script src="jquery.min.js"></script>
  <link href="select2.min.css" rel="stylesheet" />
  <script src="select2.min.js"></script>

  <div unicorn:ignore>
    <select
      id="select2-example"
      onchange="Unicorn.call('ignored-element', 'select_state', this.value, this.selectedInd
    >
      {% for state in states %}
      <option value="{{ state }}">{{ state }}</option>
      {% endfor %}
    </select>
  </div>

  <script>
    $(document).ready(function () {
      $("#select2-example").select2();
    });
  </script>
</div>
```

```
# ignored_element.py
from django_unicorn.components import UnicornView

class JsView(UnicornView):
    states = (
        "Alabama",
```

```
        "Alaska",
        "Wisconsin",
        "Wyoming",
    )
    selected_state = ""

    def select_state(self, state_name, selected_idx):
        print("select_state state_name", state_name)
        print("select_state selected_idx", selected_idx)
        self.selected_state = state_name
```

Actions

Components can also trigger methods from the templates by listening to any valid event type. The most common events would be `click`, `input`, `keydown`, `keyup`, and `mouseenter`, but MDN has a list of all of the browser event types available.

Events

An example action to call the `clear_name` method on the component.

```
<!-- clear-name.html -->
<div>
  <input unicorn:model="name" type="text" id="text" />
  Hello {{ name|title }}
  <button unicorn:click="clear_name">Clear Name</button>
</div>
```

```
# clear_name.py
from django_unicorn.components import UnicornView

class ClearNameView(UnicornView):
    name = "World"

    def clear_name(self):
        self.name = ""
```

When the button is clicked, the `name` property will get set to an empty string. Then, the component will intelligently re-render itself and the text input will update to match the property on the component.

> ***Tip***
>
> Instance methods without arguments can be called from the template with or without parenthesis.

Passing arguments

Actions can also pass basic Python types to the backend component.

```
<!-- passing-args.html -->
<div>
  <input unicorn:model="name" type="text" id="text" />
  Hello {{ name|title }} ■
  <button unicorn:click="set('Bob')">Set as Bob</button>
  <button unicorn:click="set()">Set default value of name argument</button>
</div>
```

```
# passing_args.py
from django_unicorn.components import UnicornView
```

```
class PassingArgsView(UnicornView):
    name = "World"

    def set(self, name="Universe"):
        self.name = name
```

Argument types

Most basic Python types, including `string`, `int`, `float`, `bool`, `list`, `tuple`, `dictionary`, and `set`, are supported by default.

```
<!-- argument-types.html -->
<div>
  <button unicorn:click="update(99)">Pass int</button>
  <button unicorn:click="update(1.234)">Pass float</button>
  <button unicorn:click="update(True)">Pass bool</button>
  <button unicorn:click="update({'key': 'value'})">Pass dictionary</button>
  <button unicorn:click="update([1, 2, 3])">Pass list</button>
  <button unicorn:click="update((1, 2, 3))">Pass tuple</button>
  <button unicorn:click="update({1, 2, 3})">Pass set</button>
</div>
```

Coerced types

Arguments with the types of `datetime`, `date`, `time`, `timedelta`, and `UUID` can be coerced by using a type annotation in the action method.

Note

Django's `dateparse` methods are used to convert strings to the date-related type.

Note

Both integer and float epochs can also be coerced into `datetime` or `date` objects.

```
# argument_type_hints.py
from django_unicorn.components import UnicornView
from datetime import date, datetime
from uuid import UUID

class ArgumentTypeHintsView(UnicornView):
    def is_datetime(self, obj: datetime):
        assert type(obj) is datetime

    def is_uuid(self, obj: UUID):
        assert type(obj) is UUID

    def is_date(self, _date: date = None):
        assert type(obj) is _date
```

```
<!-- argument-type-hints.html -->
<div>
  <button unicorn:click="is_datetime('2020-09-12T01:01:01')">Check datetime with string</but
  <button unicorn:click="is_datetime(1691499534)">Check datetime with epoch</button>
  <button unicorn:click="is_uuid('90144cb9-fc47-476d-b124-d543b0cff091')">Check UUID</button
```

```
  <button unicorn:click="is_date(date='2020-09-12')">Check date</button>
</div>
```

Django models

Django models can be instantiated as an argument or with a `pk` kwarg and a type annotation.

```
# argument_model.py
from django_unicorn.components import UnicornView
from django.contrib.auth.models import User

class ArgumentModelView(UnicornView):
    def is_user_via_arg(self, obj: User):
        assert type(obj) is User

    def is_user_via_kwarg(self, pk: User=None):
        assert type(obj) is User
```

```
<!-- argument-model.html -->
<div>
  <button unicorn:click="is_user_via_arg(1)">Gets user with pk of 1</button>
  <button unicorn:click="is_user_via_kwarg(pk=2)">Gets user with pk of 2</button>
</div>
```

Custom types

Custom objects can also be used as a type annotation and `Unicorn` will attempt to instantiate the object with the value that is passed in as the argument.

```
# argument_custom_type.py
from django_unicorn.components import UnicornView

class CustomType:
  def __init__(self, custom_type_id: int):
    self.custom_type_id = custom_type_id

class ArgumentCustomTypeView(UnicornView):
    def is_custom_type(self, obj: CustomType):
        assert type(obj) is CustomType
```

```
<!-- argument-custom-type.html -->
<div>
  <button unicorn:click="is_custom_type(1)">Gets custom type</button>
</div>
```

Enums

Enums themselves cannot be passed as an argument, but the enum *value* can be since that is a Python primitive. Use the enum as a type annotation to coerce the value into the specified enum.

```
# enum.py
from django_unicorn.components import UnicornView
from enum import Enum

class Color(Enum):
  RED = 1
  GREEN = 2
  BLUE = 3
  PURPLE = 4

class EnumView(UnicornView):
```

```
    color = Color.RED
    purple_color = Color.PURPLE

    def set_color(self, color: Color):
      self.color = color
```

```
<!-- enum.html -->
<div>
  <button unicorn:click="set_color({{ color.BLUE.value }})">Show BLUE (and 3) below when cli
  <button unicorn:click="set_color(2)">Show GREEN (and 2) below when clicked</button>
  <button unicorn:click="set_color({{ purple_color.value }})">Show PURPLE (and 4) below when

  <br />
  <!-- This will be RED when first rendered, and then will change based on the button clicke
  Color: {{ color }}

  <br />
  <!-- This will be 1 when first rendered, and then will change based on the button clicked
  Color int: {{ color.value }}
</div>
```

Set shortcut

Actions can also set properties without requiring an explicit method.

```
<!-- set-shortcut.html -->
<div>
  <input unicorn:model="name" type="text" id="text" />
  Hello {{ name|title }} ■
  <button unicorn:click="name='Bob'">Set name as Bob</button>
</div>
```

```
# set_shortcut.py
from django_unicorn.components import UnicornView

class SetShortcutView(UnicornView):
    name = "World"
```

Modifiers

Similar to models, actions also have modifiers which change how the method gets called.

prevent

Prevents the default action the browser would use for that element. The same as calling `preventDefault`.

```
<!-- prevent-modifier.html -->
<div>
  <button unicorn:click.prevent="name='Bob'">Set name as Bob</button>
</div>
```

stop

Stops the event from bubbling up the event chain. The same as calling `stopPropagation`.

```
<!-- stop-modifier.html -->
<div>
    <button unicorn:click.stop="name='Bob'">Set name as Bob</button>
</div>
```

discard

Discards any model updates from being saved before calling the specified method on the view. Useful for a cancel button.

```
<!-- discard-modifier.html -->
<div>
    <input type="text" unicorn:model="name">
    <button unicorn:click.discard="cancel">Cancel</button>
</div>
```

```
# discard_modifier.py
from django_unicorn.components import UnicornView

class DiscardModifierView(UnicornView):
    name = None

    def cancel(self):
        pass
```

debounce

Waits the specified time in milliseconds before calling the specified method.

```
<!-- debounce-modifier.html -->
<div>
    <input type="text" unicorn:model="name">
    <button unicorn:click.debounce-1000="add_count">Add Count</button>
</div>
```

Special arguments

$event

A reference to the event that triggered the action.

```
<!-- event.html -->
<div>
    <input type="text" unicorn:change="update($event.target.value.trim())">Update</input>
</div>
```

$returnValue

A reference to the last return value from an action method.

```
<!-- returnValue.html -->
<div>
    <input type="text" unicorn:change="update($returnValue.trim())">Update</input>
</div>
```

Special methods

$refresh

Refresh and re-render the component from its current state.

```
<!-- refresh-method.html -->
<div>
  <button unicorn:click="$refresh">Refresh the component</button>
</div>
```

$reset

Revert the component to its original state.

```
<!-- reset-method.html -->
<div>
  <button unicorn:click="$reset">Reset the component</button>
</div>
```

$toggle

Toggle a field on the component. Can only be used for fields that are booleans.

```
<!-- toggle-method.html -->
<div>
  <button unicorn:click="$toggle('check')">Toggle the check field</button>
</div>
```

> ***Tip***
>
> Multiple fields can be toggled at the same time by passing in multiple fields at a time: `unicorn:click="$toggle('check', 'another_check', 'a_third_check')"`. Nested properties are also supported: `unicorn:click="$toggle('nested.check')"`.

$validate

Validates the component.

```
<!-- validate-method.html -->
<div>
  <button unicorn:click="$validate">Validate the component</button>
</div>
```

Calling methods

Sometimes you need to trigger a method on a component from regular JavaScript. That is possible with `Unicorn.call()`. The first argument is the name (or key) of the component and the second argument is the name of the method to call.

```
<!-- call-with-component-name.html -->
{% unicorn 'hello-world' %}

<button onclick="Unicorn.call('hello-world', 'set_name');">
  Set the name from outside the component
</button>
```

```
<!-- call-with-component-key.html -->
{% unicorn 'hello-world' key='hello-universe' %}

<button onclick="Unicorn.call('hello-universe', 'set_name');">
  Set the name from outside the component
</button>
```

Passing arguments to the method call is also supported.

```
<!-- index.html -->
{% unicorn 'hello-world' %}

<button onclick="Unicorn.call('hello-world', 'set_name', 'World');">
```

```
  Set the name to "World" from outside the component
</button>
```

Return values

To retrieve the last action method's return value, use `Unicorn.getReturnValue()`.

```
<!-- index.html -->
{% unicorn 'hello-world' %}

<button onclick="alert(Unicorn.getReturnValue('hello-world'));">
  Get the last return value
</button>
```

Child components

`Unicorn` supports nesting components so that one component is a child of another. Since HTML is a tree structure, a component can have multiple children, but each child only has one parent.

We will slowly build a nested component example with three components: table, filter and row. The table is the parent and contains the other two components. The filter will update the parent table component, and the row will contain functionality to edit a row.

Parent component

So that `Unicorn` knows about the parent-child relationship, the child component must pass in a `parent` keyword argument with the parent's component view.

```
<!-- table.html -->
{% load unicorn %}
<div>
  {% unicorn 'filter' parent=view %}

  <table>
    <thead>
      <tr>
        <td>Author</td>
        <td>Title</td>
      </tr>
    </thead>
    {% for book in books %}
    <tr>
      <td>{{ book.author }}</td>
      <td>{{ book.title }}</td>
    </tr>
    {% endfor %}
  </table>
</div>
```

```
# table.py
from book.models import Book
from django_unicorn.components import UnicornView

class TableView(UnicornView):
    books = Book.objects.none()

    def mount(self):
        self.load_table()
```

```
    def load_table(self):
        self.books = Book.objects.all()[0:10]
```

Note

`view` will always be the current component's view, so passing `view` into `parent` (i.e. `parent=view`) will always create the relationship correctly.

Child component

The child filter component, `{% unicorn 'filter' parent=view %}`, will have access to its parent through the view's `self.parent`. The `FilterView` is using the updated method to filter the `books` queryset on the table component when the filter's `search` model is changed.

```
<!-- filter.html -->
<div>
  <input type="text" unicorn:model="search" />
</div>
```

```
from django_unicorn.components import UnicornView

class FilterView(UnicornView):
    search = ""

    def updated_search(self, query):
        self.parent.load_table()

        if query:
            self.parent.books = list(
                filter(lambda f: query.lower() in f.title.lower(), self.parent.books)
            )
```

Multiple children

If we want to encapsulate the editing and saving of a row of data, we can add in a row component as well.

Note

The discard action modifier is used on the cancel button to provide an easy way to prevent any edits from being saved.

```
<!-- row.html -->
<tr>
  <td>
    {% if is_editing %}
    <input type="text" unicorn:model.defer="book.author" />
    {% else %}
    {{ book.author }}
    {% endif %}
  </td>
  <td>
    {% if is_editing %}
    <input type="text" unicorn:model.defer="book.title" />
    {% else %}
    {{ book.title }}
```

```
    {% endif %}
  </td>
  <td>
    {% if is_editing %}
    <button unicorn:click="save">Save</button>
    <button unicorn:click.discard="cancel">Cancel</button>
    {% else %}
    <button unicorn:click="edit">Edit</button>
    {% endif %}
  </td>
</tr>
```

```
# row.py
from django_unicorn.components import UnicornView

class RowView(UnicornView):
    book = None
    is_editing = False

    def edit(self):
        self.is_editing = True

    def cancel(self):
        self.is_editing = False

    def save(self):
        self.book.save()
        self.is_editing = False
```

The changes for the table component where the row child component is added in. Views will always have a `children` attribute – here it is used to set `is_editing` to `false` on all rows when the table gets reloaded.

```
<!-- table.html --->
{% load unicorn %}
<div>
  {% unicorn 'filter' parent=view %}

  <table>
    <thead>
      <tr>
        <td>Author</td>
        <td>Title</td>
      </tr>
    </thead>
    {% for book in books %} {% unicorn 'row' parent=view book=book key=book.id
    %} {% endfor %}
  </table>
</div>
```

```
# table.py
from book.models import Book
from django_unicorn.components import UnicornView

class TableView(UnicornView):
    books = Book.objects.none()

    def mount(self):
        self.load_table()

    def load_table(self):
```

```
        self.books = Book.objects.all()

        for child in self.children:
            if hasattr(child, "is_editing"):
                child.is_editing = False
```

Warning

`Unicorn` requires components to have a unique identifier. Normally that is handled automatically, however multiple child components with the same component name require some help.

For child components, `unicorn:id` is automatically created from the parent's `unicorn:id` and the child's component name. If a child component is created multiple times in the same parent, one of the following can be used to create unique identifiers:

- pass in a `key` keyword argument to the child component

```
{% unicorn 'row' parent=view book=book key=book.id %}
```

- pass in an `id` keyword argument to the child component

```
{% unicorn 'row' parent=view book=book id=book.id %}
```

- the view has an attribute named `model` which has either a `pk` or `id` attribute

```
{% unicorn 'row' parent=view model=book %}
```

Django Models

`Unicorn` provides tight integration with Django `Models` and `QuerySets` to handle typical workflows.

Model

A Django `Model` can be used as a field on a component just like basic Python primitive types. Use `unicorn:model` to bind to a field of a Django `Model` like you would in a normal Django template.

Warning

Using this functionality will serialize your entire model by default and expose all of the values in the HTML source code. Do not use this particular functionality if there are properties that need to be kept private.

One option is to customize the serialization of the model into a dictionary to only expose the data that should be publicly available.

Another option is to use Meta.exclude or Meta.javascript_exclude so those fields are not exposed.

```
<!-- model.html -->
<div>
  <input unicorn:model.defer="book.title" type="text" id="book" />
  {{ book.title }}
  <button unicorn:click="save({{ book.pk }})">Save</button>
</div>
```

```
# model.py
from django_unicorn.components import UnicornView
from books.models import Book
```

```
class ModelView(UnicornView):
    book: Book = None

    def mount(self):
      self.book = Book.objects.all().first()

    def save(self, book_to_save: Book):
        book_to_save.save()
```

Note

The model's `pk` will be used to look up the correct model if there is only one argument for an action method and it has a type annotation for a Django `Model`. To lookup by a different model field, pass a dictionary into the front-end.

```
<button unicorn:click="delete({ 'uuid': '{{ book.uuid }}'})">Delete by uuid</button>
```

```
def delete(self, book_to_delete: Book):
    book_to_delete.delete()
```

QuerySet

A Django `QuerySet` can be set to a property on a component just like a regular `list`.

```
<!-- queryset.html -->
<div>
  {% for book in books %}
  <div>
    <div>
      <input unicorn:model.defer="books.{{ forloop.counter0 }}.title" type="text" id="title"
      {{ book.title }}
    </div>
    <div>
      <input unicorn:model.defer="books.{{ forloop.counter0 }}.description" type="text" id="
      {{ book.description }}
    </div>
    <div>
      <button unicorn:click="save({{ forloop.counter0 }})">Save</button>
    </div>
  </div>
  {% endfor %}
</div>
```

```
# queryset.py
from django_unicorn.components import UnicornView
from books.models import Book

class QuerysetView(UnicornView):
    books = Book.objects.none()

    def mount(self):
        self.books = Book.objects.all().order_by("-id")[:5]

    def save(self, book_idx: int):
        self.books[book_idx].save()
```

Warning

This will expose all of the model values for the `QuerySet` in the HTML source. One way to avoid leaking all model information is to pass the fields that are publicly viewable into `values()` on your `QuerySet`.

```
def mount(self):
  self.books = Book.objects.all().order_by("-id").values("pk", "title")[:5]
```

A `QuerySetType` type hint can also be used for `QuerySet` to ensure the correct type is used for the component field.

```
# queryset.py
from django_unicorn.components import QuerySetType, UnicornView
from books.models import Book

class QuerysetView(UnicornView):
    books: QuerySetType[Book] = None

    def mount(self):
        self.books = Book.objects.all().order_by("-id")[:5]

    def save(self, book_idx: int):
        self.books[book_idx].save()
```

Direct View

Usually components will be included in a regular Django template, however a component can also be specified in a `urls.py` file in instances where the having an additional template is not necessary.

Template Requirements

- there must be one (and only one) element that wraps around the portion of the template that should be handled by `Unicorn`
- the wrapping element must include `unicorn:view` as an attribute
- the template must included the `unicorn_scripts` and `csrf_token` template tags

Similar to a class-based view, `Unicorn` components have a `as_view` function which is used in `urls.py`.

Example

```
# book.py
from django_unicorn.components import UnicornView

class BookView(UnicornView):
    title = ""
```

```
<!-- book.html -->
{% load unicorn %}

<html>
  <head>
    {% unicorn_scripts %}
  </head>
  <body>
    {% csrf_token %}
    <h1>Book</h1>
```

```
    <div unicorn:view>
      <input unicorn:model="title" type="text" /><br />
      {{ title }}
    </div>
  </body>
</html>
```

```
# urls.py
from django.urls import path
from unicorn.components.book import BookView

urlpatterns = [
    path("book", BookView.as_view(), name="book"),
]
```

Validation

`Unicorn` has two options for validation. It can either use the standard Django `forms` infrastructure for re-usability or `ValidationError` can be raised for simpler use-cases.

ValidationError

If you do not want to create a form class or you want to specifically target a nested field you can raise a `ValidationError` inside of an action method. The `ValidationError` must be instantiated with a `dict` with the model name as the key and error message as the value. A `code` keyword argument must also be passed in. The typical error codes used are `required` or `invalid`.

```
# book_validation_error.py
from django.utils.timezone import now
from django_unicorn.components import UnicornView

class BookView(UnicornView):
    book: Book

    def publish(self):
        if not self.book.title:
            raise ValidationError({"book.title": "Books must have a title"}, code="required"

        self.publish_date = now()
        self.book.save()
```

```
<!-- book-validation-error.html -->
<div>
  <input unicorn:model="book.title" type="text" id="title" /><br />
  <button unicorn:click="publish">Publish Book</button>
</div>
```

Django Form

`Unicorn` can use the Django `forms` infrastructure for validation. This means that a form could be re-used between any other Django views and a `Unicorn` component.

Note

There are many built-in fields available for Django form fields which can be used to validate text inputs.

```python
# book_form.py
from django_unicorn.components import UnicornView
from django import forms

class BookForm(forms.Form):
    title = forms.CharField(max_length=100, required=True)
    publish_date = forms.DateField(required=True)

class BookView(UnicornView):
    form_class = BookForm

    title = ""
    publish_date = ""
```

```html
<!-- book-form.html -->
<div>
  <input unicorn:model="title" type="text" id="title" /><br />
  <input unicorn:model="publish_date" type="text" id="publish-date" /><br />
  <button unicorn:click="$validate">Validate</button>
</div>
```

Because of the `form_class = BookForm` defined on the `UnicornView` above, `Unicorn` will automatically validate that the title has a value and is less than 100 characters. The `publish_date` will also be converted into a `datetime` from the string representation in the text input.

Validate the entire component

The magic action method `$validate` can be used to validate the whole component using the specified form.

```html
<!-- validate.html -->
<div>
  <input unicorn:model="publish_date" type="text" id="publish-date" /><br />
  <button unicorn:click="$validate">Validate</button>
</div>
```

The `validate` method can also be used inside of the component.

```python
# validate.py
from django_unicorn.components import UnicornView
from django import forms

class BookForm(forms.Form):
    title = forms.CharField(max_length=6, required=True)

class BookView(UnicornView):
    form_class = BookForm

    text = "hello"

    def set_text(self):
        self.text = "hello world"
        self.validate()
```

The `is_valid` method can also be used inside of the component to check if a component is valid.

```python
# validate.py
from django_unicorn.components import UnicornView
from django import forms

class BookForm(forms.Form):
    title = forms.CharField(max_length=6, required=True)
```

```
class BookView(UnicornView):
    form_class = BookForm

    text = "hello"

    def set_text(self):
        if self.is_valid():
            self.text = "hello world"
```

Showing validation errors

There are a few ways to show the validation messages.

Highlighting the invalid form

When a model form is invalid, a special `unicorn:error` attribute is added to the element. Depending on whether it is an `invalid` or `required` error code, the attribute will be `unicorn:error:invalid` or `unicorn:error:required`. The value of the attribute will be the validation message.

```
<!-- highlight-input-errors.html -->
<div>
  <style>
    [unicorn\:error\:invalid] {
      border: 1px solid red !important;
    }
    [unicorn\:error\:required] {
      border: 1px solid red !important;
    }
  </style>

<input
  unicorn:model="publish_date"
  type="text"
  id="publish-date"
  unicorn:error:invalid="Enter a valid date/time."
/><br />

</div>
```

Showing a specific error message

```
<!-- show-error-message.html -->
<div>
  <input unicorn:model="publish_date" type="text" id="publish-date" /><br />
  <span class="error">{{ unicorn.errors.publish_date.0.message }}</span>
</div>
```

Showing all the error messages

There is a `unicorn_errors` template tag that shows all errors for the component. It provides an example of how to display component errors in a more specific way if needed.

```
<!-- show-all-error-messages.html -->
{% load unicorn %}

<div>
  {% unicorn_errors %}

  <input unicorn:model="publish_date" type="text" id="publish-date" /><br />
</div>
```

Redirecting

`Unicorn` has a few different ways to redirect from an action method.

Redirect

To redirect the user, return a `HttpResponseRedirect` from an action method. Using the Django shortcut `redirect` method is one way to do that in a typical Django manner.

Note

`django.shortcuts.redirect` can take a Django model, Django view name, an absolute url, or a relative url. However, the `permanent` kwarg for `redirect` has no bearing in this context.

Tip

It is not required to use `django.shortcuts.redirect`. Anything that returns a `HttpResponseRedirect` will behave the same in `Unicorn`.

```python
# redirect.py
from django.shortcuts import redirect
from django_unicorn.components import UnicornView
from .models import Book

class BookView(UnicornView):
    title = ""

    def save_book(self):
        book = Book(title=self.title)
        book.save()
        self.reset()

        return redirect(f"/book/{book.id}")
```

```html
<!-- redirect.html -->
<div>
  <input unicorn:model="title" type="text" id="title" /><br />
  <button unicorn:click="save_book()">Save book</button>
</div>
```

HashUpdate

To avoid a server-side page refresh and just update the hash at the end of the url, return `HashUpdate` from the action method.

```python
# hash_update.py
from django_unicorn.components import HashUpdate, UnicornView
from .models import Book

class BookView(UnicornView):
    title = ""

    def save_book(self):
        book = Book(title=self.title)
        book.save()
        self.reset()
```

```
        return HashUpdate(f"#{book.id}")
```

```
<!-- hash-update.html -->
<div>
  <input unicorn:model="title" type="text" id="title" /><br />
  <button unicorn:click="save_book()">Save book</button>
</div>
```

LocationUpdate

To avoid a server-side page refresh and update the whole url, return a `LocationUpdate` from the action method.

`LocationUpdate` is instantiated with a `HttpResponseRedirect` arg and an optional `title` kwarg.

> ***Note***
>
> `LocationUpdate` uses `window.history.pushState` so the new url must be relative or the same origin as the original url.

```
# location_update.py
from django.shortcuts import redirect
from django_unicorn.components import LocationUpdate, UnicornView
from .models import Book

class BookView(UnicornView):
    title = ""

    def save_book(self):
        book = Book(title=self.title)
        book.save()
        self.reset()

        return LocationUpdate(redirect(f"/book/{book.id}"), title=f"{book.title}")
```

```
<!-- location-update.html -->
<div>
  <input unicorn:model="title" type="text" id="title" /><br />
  <button unicorn:click="save_book()">Save book</button>
</div>
```

Loading States

`Unicorn` requires an AJAX request for any component updates, so it is helpful to provide some context to the user that an action is happening.

Toggling Elements

Elements with the `unicorn:loading` attribute are only visible when an action is in process.

```
<!-- loading.html -->
<div>
  <button unicorn:click="update">Update</button>

  <div unicorn:loading>Updating!</div>
</div>
```

When the *Update* button is clicked, the “Updating!” message will show until the action is complete, and then it will re-hide itself.

Warning

Loading elements get shown or removed with the `hidden` attribute. One drawback to this approach is that setting the style `display` property overrides this functionality.

You can also hide an element while an action is processed by adding a `remove` modifier.

```
<!-- loading-remove.html -->
<div>
  <button unicorn:click="update">Update</button>

  <div unicorn:loading.remove>Not currently updating!</div>
</div>
```

If there are multiple actions that happen in the component, you can show or hide a loading element for a specific action by targeting another element’s `id` with `unicorn:target`.

```
<!-- loading-target-id.html -->
<div>
  <button unicorn:click="update" id="updateId">Update</button>
  <button unicorn:click="delete" id="deleteId">Delete</button>

  <div unicorn:loading unicorn:target="updateId">Updating!</div>
  <div unicorn:loading unicorn:target="deleteId">Deleting!</div>
</div>
```

An element’s `unicorn:key` can also be targeted.

```
<!-- loading-target-key.html -->
<div>
  <button unicorn:click="update" unicorn:key="updateKey">Update</button>
  <button unicorn:click="delete" unicorn:key="deleteKey">Delete</button>

  <div unicorn:loading unicorn:target="updateKey">Updating!</div>
  <div unicorn:loading unicorn:target="deleteKey">Deleting!</div>
</div>
```

Note

An asterisk can be used as wildcard to target more than one element at a time.

```
<!-- loading-target-wildcard-id.html -->
<div>
  <button unicorn:click="update" id="update1Id">Update 1</button>
  <button unicorn:click="update" id="update2Id">Update 2</button>

  <div unicorn:loading unicorn:target="update*Id">Updating!</div>
</div>
```

Toggling Attributes

Elements with an action event can also include an `unicorn:loading` attribute with either an `attr` or `class` modifier.

attr

Set the specified attribute on the element that is triggering the action.

This example will disable the *Update* button when it is clicked and remove the attribute once the action is completed.

```
<!-- loading-attr.html -->
<div>
  <button unicorn:click="update" unicorn:loading.attr="disabled">Update</button>
</div>
```

class

Add the specified class(es) to the element that is triggering the action.

This example will add `loading` and `spinner` classes to the *Update* button when it is clicked and remove the classes once the action is completed.

```
<!-- loading-class.html -->
<div>
  <button unicorn:click="update" unicorn:loading.class="loading spinner">Update</button>
</div>
```

class.remove

Remove the specified class from the element that is triggering the action.

This example will remove a `active` class from the *Update* button when it is clicked and add the class back once the action is completed.

```
<!-- loading-class-remove.html -->
<div>
  <button unicorn:click="update" unicorn:loading.class.remove="active">
    Update
  </button>
</div>
```

Dirty States

`Unicorn` can provide context to the user that some data has been changed and will be updated.

Toggling Attributes

Elements can include an `unicorn:dirty` attribute with either an `attr` or `class` modifier.

attr

Set the specified attribute on the element that is changed.

This example will set the input to be readonly when the model is changed. The attribute will be removed once the name is synced or if the input value is changed back to the original value.

```
<!-- dirty-attr.html -->
<div>
  <input unicorn:model="name" unicorn:dirty.attr="readonly" />
</div>
```

class

Add the specified class(es) to the model that is changed.

This example will add *dirty* and *changing* classes to the input when the model is changed. The classes will be removed once the model is synced or if the input value is changed back to the original value.

```
<!-- dirty-class.html -->
<div>
  <input unicorn:model="name" unicorn:dirty.class="dirty changing" />
</div>
```

class.remove

Remove the specified class(es) from the model that is changed.

This example will remove the *clean* class from the input when the model is changed. The class will be added back once the model is synced or if the input value is changed back to the original value.

```
<!-- dirty-class-remove.html -->
<div>
  <input unicorn:model="name" unicorn:dirty.class.remove="clean" />
</div>
```

Partial Updates

Normally `Unicorn` will send the entire component's rendered HTML on every action to make sure that any changes to the context is reflected on the page. However, to reduce latency and minimize the amount of data that has to be sent over the network, `Unicorn` can only update a portion of the page by utilizing the `unicorn:partial` attribute.

> ***Note***
>
> By default, `unicorn:partial` will look in the current component's template for an `id` or `unicorn:key`. If an element can't be found with the specified target, the entire component will be morphed like usual.

```
# partial_update.py
from django_unicorn.components import UnicornView

class PartialUpdateView(UnicornView):
    checked = False
```

```
<!-- partial-update.html -->
<div>
  <span id="checked-id">{{ checked }}</span>
  <button unicorn:click="$toggle('checked')" unicorn:partial="checked-id">
    Toggle checked
  </button>
</div>
```

Target by id

To only target an element `id` add the `id` modifier to `unicorn:partial`.

```
<!-- partial-update-id.html -->
<div>
  <span id="checked-id">{{ checked }}</span>
  <button unicorn:click="$toggle('checked')" unicorn:partial.id="checked-id">
    Toggle checked
  </button>
</div>
```

Target by key

To only target an element `unicorn:key` add the `key` modifier to `unicorn:partial`.

```
<!-- partial-update-key.html -->
<div>
  <span unicorn:key="checked-key">{{ checked }}</span>
  <button unicorn:click="$toggle('checked')" unicorn:partial.key="checked-key">
    Toggle checked
  </button>
</div>
```

e

e partials can be targetted by adding multiple attributes to the element.

```
ton unicorn:click="$toggle('checked')" unicorn:partial.key="checked-key" unicorn:partial.id="checke
```

Polling

`unicorn:poll` can be added to the root `div` element of a component to have it refresh the component automatically every 2 seconds. The polling is smart enough that it won't poll when the page is inactive.

```
# polling.py
from django.utils.timezone import now
from django_unicorn.components import UnicornView

class PollingView(UnicornView):
    current_time = now()
```

```
<!-- polling.html -->
<div unicorn:poll>{{ current_time }}</div>
```

A method can also be specified if there is a specific method on the component that should called every time the polling fires. For example, `unicorn:poll="get_updates"` would call the `get_updates` method instead of the built-in `refresh` method.

To define a different refresh time in milliseconds, a modifier can be added as well. `unicorn:poll-1000` would fire the `refresh` method every 1 second, instead of the default 2 seconds.

```
<!-- polling-every-five-seconds.html -->
<div unicorn:poll-5000="get_updates">
  <input unicorn:model="update" type="text" id="text" />
  {{ update }}
</div>
```

Disable poll

Polling can dynamically be disabled by checking a boolean field from the component.

```
# poll_disable.py
from django.utils.timezone import now
from django_unicorn.components import UnicornView

class PollDisableView(UnicornView):
    polling_disabled = False
    current_time = now()

    def get_date(self):
        self.current_time = now()
```

```
<!-- poll-disable.html -->
<div unicorn:poll-1000="get_date" unicorn:poll.disable="polling_disabled">
    current_time: {{ current_time|date:"s" }}<br />
    <button u:click="$toggle('polling_disabled')">Toggle Polling</button>
</div>
```

Note

The field passed into `unicorn:poll.disable` can be negated with an exclamation point.

```
# poll_disable_negation.py
from django.utils.timezone import now
from django_unicorn.components import UnicornView

class PollDisableNegationView(UnicornView):
    polling_enabled = True
    current_time = now()

    def get_date(self):
        self.current_time = now()
```

```
<!-- poll-disable-negation.html -->
<div unicorn:poll-1000="get_date" unicorn:poll.disable="!polling_enabled">
    current_time: {{ current_time|date:"s" }}<br />
    <button u:click="$toggle('polling_enabled')">Toggle Polling</button>
</div>
```

PollUpdate

A poll can be dynamically updated by returning a `PollUpdate` object from an action method. The timing and method can be updated, or it can be disabled.

```
# poll_update.py
from django.utils.timezone import now
from django_unicorn.components import PollUpdate, UnicornView

class PollingUpdateView(UnicornView):
    polling_disabled = False
    current_time = now()

    def get_date(self):
        self.current_time = now()
        return PollUpdate(timing=2000, disable=False, method="get_date")
```

```
<!-- poll-update.html -->
<div unicorn:poll-1000="get_date">
    current_time: {{ current_time|date:"s" }}<br />
</div>
```

Visibility

`unicorn:visible` can be added to any element to have it call the specified view method when it scrolls into view.

```
# visibility.py
from django_unicorn.components import UnicornView

class VisibilityView(UnicornView):
```

```
    visibility_count = 0

    def add_count(self):
        self.visibility_count += 1
```

```
<!-- visibility.html -->
<div>
    <div style="height: 100%">
        <span unicorn:visible="add_count">
    </div>
</div>
```

Note

In some cases, the element with the `unicorn:visible` attribute will always be in the viewport, so the event will continue to fire and the method will continue to execute. However, this will not happen in the following instances:

- the fields of component do not change, so the AJAX request will return a 304 status code
- the method explicitly returns `False`

Modifiers

There are a few modifiers for `unicorn:visible` and they are able to be chained if necessary.

Debounce

Similar to the debounce modifier on a model or actions, wait the specified number of milliseconds before calling the specified method.

```
<!-- debounce-modifier.html -->
<div>
    <div style="height: 100%">
        <span unicorn:visible.debounce-1000="add_count"></span>
    </div>
</div>
```

Threshold

The percentage (as an integer) that should be visible before being triggered. For example, `0` means that as soon as 1 pixel of the element is visible it would be fired, `25` would be 25% of the target element is visible, `100` would require 100% of the element to be completely visible.

```
<!-- threshold-modifier.html -->
<div>
    <div style="height: 100%">
        <span unicorn:visible.threshold-25="add_count"></span>
    </div>
</div>
```

Messages

`Unicorn` supports Django messages and they work the same as if the template was rendered server-side. When the `update` action is fired, a success message will be added to the request and will show up inside the component.

```
<!-- messages.html -->
<div>
    {% if messages %}
```

```
    <ul class="messages">
        {% for message in messages %}
        <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{ message }}</li>
        {% endfor %}
    </ul>
    {% endif %}

    <button unicorn:click="update">Update</button>

</div>
```

```
# messages.py
from django.contrib import messages
from django_unicorn.components import UnicornView

class MessagesView(UnicornView):
    def update(self):
        messages.success(self.request, "update called")
```

Redirecting

When the action returns a `redirect`, `Unicorn` will defer the messages so they do not get rendered in the component (since the user will never see the re-rendered component). Once the redirect has happened `messages` will be available for rendering by the template as expected.

```
<!-- messages-when-redirecting.html -->
<div>
    <button unicorn:click="update">Update</button>
</div>
```

```
<!-- new-url.html -->

{% if messages %}

<ul class="messages">
    {% for message in messages %}
    <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

```
# messages_when_redirecting.py
from django.contrib import messages
from django.shortcuts import redirect
from django_unicorn.components import UnicornView

class MessagesWhenRedirectingView(UnicornView):
    def update(self):
        messages.success(self.request, "update called")

        return redirect("new-url")
```

Advanced Views

Class properties

template_name

By default, the component name is used to determine what template should be used. For example, `hello_world.HelloWorldView` would by default use `unicorn/hello-world.html`. However, you can specify a particular template by setting `template_name` in the component.

```
# hello_world.py
from django_unicorn.components import UnicornView

class HelloWorldView(UnicornView):
    template_name = "unicorn/hello-world.html"
```

Instance properties

component_args

The arguments passed into the component.

```
<!-- index.html -->
{% unicorn 'hello-arg' 'World' %}
```

```
# hello_arg.py
from django_unicorn.components import UnicornView

class HelloArgView(UnicornView):
    def mount(self):
      assert self.component_args[0] == "World"
```

component_kwargs

The keyword arguments passed into the component.

```
<!-- index.html -->
{% unicorn 'hello-kwarg' hello='World' %}
```

```
# hello_kwarg.py
from django_unicorn.components import UnicornView

class HelloKwargView(UnicornView):
    def mount(self):
      assert self.component_kwargs["hello"] == "World"
```

request

The current `request`.

```
# hello_world.py
from django_unicorn.components import UnicornView

class HelloWorldView(UnicornView):
    def mount(self):
        print("Initial request that rendered the component", self.request)

    def test(self):
        print("AJAX request that re-renders the component", self.request)
```

Custom methods

Defined component instance methods with no arguments are made available to the Django template context and can be called like a property.

```
# states.py
from django_unicorn.components import UnicornView

class StateView(UnicornView):
    def all_states(self):
        return ["Alabama", "Alaska", "Arizona", ...]
```

```
<!-- states.html -->
<div>
  <ul>
    {% for state in all_states %}
    <li>{{ state }}</li>
    {% endfor %}
  </ul>
</div>
{% endverbatim %}
```

Tip

If the method is intensive and will be called multiple times, it can be cached with Django's cached_property to prevent duplicate API requests or database queries. The method will only be executed once per component rendering.

```
# states.py
from django.utils.functional import cached_property
from django_unicorn.components import UnicornView

class StateView(UnicornView):
    @cached_property
    def all_states(self):
        return ["Alabama", "Alaska", "Arizona", ...]
```

Instance methods

mount()

Gets called when the component gets initialized or reset.

```
# hello_world.py
from django_unicorn.components import UnicornView

class HelloWorldView(UnicornView):
    name = "original"

    def mount(self):
        self.name = "mounted"
```

hydrate()

Gets called when the component data gets set.

```
# hello_world.py
from django_unicorn.components import UnicornView
```

```
class HelloWorldView(UnicornView):
    name = "original"

    def hydrate(self):
        self.name = "hydrated"
```

updating(name, value)

Gets called before each property that will get set.

updated(name, value)

Gets called after each property gets set.

updating_{property_name}(value)

Gets called before the specified property gets set.

updated_{property_name}(value)

Gets called after the specified property gets set.

calling(name, args)

Gets called before each method that gets called.

called(name, args)

Gets called after each method gets called.

complete()

Gets called after all methods have been called.

rendered(html)

Gets called after the component has been rendered.

parent_rendered(html)

Gets called after the component's parent has been rendered (if applicable).

Meta

Classes that derive from `UnicornView` can include a `Meta` class that provides some advanced options for the component.

exclude

By default, all public attributes of the component are included in the context of the Django template and available to JavaScript. One way to protect internal-only data is to prefix the atteibute name with _ to indicate it should stay private.

```
# hello_state.py
from django_unicorn.components import UnicornView

class HelloStateView(UnicornView):
    _all_states = (
        "Alabama",
```

```
        "Alaska",
        ...
        "Wisconsin",
        "Wyoming",
    )
```

Another way to prevent that data from being available to the component template is to add it to the `Meta` class's `exclude` tuple.

```
# hello_state.py
from django_unicorn.components import UnicornView

class HelloStateView(UnicornView):
    all_states = (
        "Alabama",
        "Alaska",
        ...
        "Wisconsin",
        "Wyoming",
    )

    class Meta:
        exclude = ("all_states", )
```

javascript_exclude

To allow an attribute to be included in the the context to be used by a Django template, but not exposed to JavaScript, add it to the `Meta` class's `javascript_exclude` tuple.

```
<!-- hello-state.html -->
<div>
  {% for state in all_states %}
  <div>{{ state }}</div>
  {% endfor %}
</div>
```

```
# hello_state.py
from django_unicorn.components import UnicornView

class HelloStateView(UnicornView):
    all_states = (
        "Alabama",
        "Alaska",
        ...
        "Wisconsin",
        "Wyoming",
    )

    class Meta:
        javascript_exclude = ("all_states", )
```

safe

By default, `unicorn` HTML encodes updated field values to prevent XSS attacks. You need to explicitly opt-in to allow a field to be returned without being encoded by adding it to the `Meta` class's `safe` tuple.

```
<!-- safe-example.html -->
<div>
  <input unicorn:model="something_safe" />
  {{ something_safe }}
</div>
```

```
# safe_example.py
from django_unicorn.components import UnicornView

class SafeExampleView(UnicornView):
    something_safe = ""

    class Meta:
        safe = ("something_safe", )
```

Note

A context variable can be marked as `safe` in the template with the normal Django template filter, as well.

```
<!-- safe-example.html -->
<div>
  <input unicorn:model="something_safe" />
  {{ something_safe|safe }}
</div>
```

JavaScript Integration

Call JavaScript from View

To integrate with other JavaScript functions, view methods can call an arbitrary JavaScript function after it gets rendered.

```
<!-- call-javascript.html -->
<div>
  <script>
    function hello(name) {
      alert("Hello, " + name);
    }
  </script>

  <input type="text" unicorn:model="name" />
  <button type="submit" unicorn:click="hello">Hello!</button>
</div>
```

```
# call_javascript.py
from django_unicorn.components import UnicornView

class CallJavascriptView(UnicornView):
    name = ""

    def mount(self):
        self.call("hello", "world")

    def hello(self):
        self.call("hello", self.name)
```

Trigger Model Update

Normally when a model element gets changed by a user it will trigger an event which `Unicorn` listens for (either `input` or `blur` depending on if it has a `lazy` modifier). However, when setting an element with JavaScript those events do not fire. `Unicorn.trigger()` provides a way to trigger that event from JavaScript manually.

The first argument to `trigger` is the component name. The second argument is the value for the element's `unicorn:key`.

```
<!-- trigger-model.html -->
<input
  id="nameId"
  unicorn:key="nameKey"
  unicorn:model="name"
  value="initial value"
/>

<script>
  document.getElementById("nameId").value = "new value";
  Unicorn.trigger("hello_world", "nameKey");
</script>
```

Queue Requests

This is an experimental feature of that queues up slow-processing component views to prevent race conditions. For simple components this should not be necessary.

Serialization is turned off by default, but can be enabled in the settings.

Warning

This feature will be disabled automatically if the cache backend is set to "django.core.cache.backends.dummy.DummyCache".

Local memory caching (the default if no `CACHES` setting is provided) will work fine if the web server only has one process. For more production use cases, consider using `redis`, `Memcache`, or database caching.

CLI

`Unicorn` provides a Django management command to create new components. The first argument is the name of the Django app to create components in. Every argument after is the name of components that should be created.

```
python manage.py startunicorn unicorn hello-world
```

This example would create a `unicorn` directory, and `templates` and `components` sub-directories if necessary. Underneath the `components` directory there will be a new module and subclass of `django_unicorn.components.UnicornView`. Underneath the `templates/unicorn` directory will be a example template.

The following is an example folder structure.

```
unicorn/
    components/
        __init__.py
        hello_world.py
    templates/
        unicorn/
            hello-world.html
```

Note

If you have an existing Django app, you can use that instead of `unicorn` like the example above. The management command will create the the directories and files as needed.

Sub-folders

`startunicorn` supports creating components in sub-folders. Separate each folder by a dot (similar to Python modules) to create a nested structure.

```
python manage.py startunicorn unicorn hello.world
```

```
unicorn/
    components/
        __init__.py
        hello/
            __init__.py
            world.py
    templates/
        unicorn/
            hello/
                world.html
```

The nested component would be included in a template like:

```
{% unicorn 'hello.world' %}
```

Settings

`Unicorn` stores all settings in a dictionary under the `UNICORN` attribute in the Django settings file. All settings are optional.

```
# settings.py
UNICORN = {
    "APPS": ["unicorn",],
    "CACHE_ALIAS": "default",
    "MINIFY_HTML": False,
    "MINIFIED": True,
    "RELOAD_SCRIPT_ELEMENTS": False,
    "SERIAL": {
        "ENABLED": False,
        "TIMEOUT": 60,
    },
    "SCRIPT_LOCATION": "after",
}
```

APPS

Specify the modules to look for components. Defaults to `["unicorn",]`.

CACHE_ALIAS

The alias to use for caching. Only used by the experimental serialization of requests for now. Defaults to `"default"`.

MINIFY_HTML

Minify the HTML generated by `Unicorn` in the AJAX request. If set to `True` and `htmlmin` is installed HTML will be minified. `htmlmin` can be installed with `Unicorn` via `poetry add django-unicorn[minify]` or `pip install django-unicorn[minify]`. Defaults to `False`.

MINIFIED

Provides a way to control if the minified version of the JavaScript bundle (i.e. `unicorn.min.js`) is used. Defaults to `!DEBUG`.

RELOAD_SCRIPT_ELEMENTS

Whether or not `script` elements in a template should be “re-run” after a template has been re-rendered.

SERIAL

Settings for the experimental serialization of requests. Defaults to `{}`.

ENABLED

Whether slow requests to the same component should be queued or not. Defaults to `False`.

TIMEOUT

The number of seconds to wait for a request to finish for additional requests to queue behind it. Defaults to `60`.

SCRIPT_LOCATION

Where the initial JavaScript data is included on initial render. Two values are currently supported: `after` and `append`.

after is the default and will render the JavaScript *outside* of the HTML component, i.e. it will be output in the same hierarchy as the parent of the HTML component.

append will render the JavaScript *inside* of the HTML component.

FAQ

Do I need to learn a new frontend framework for `Unicorn`*?*

Nope! `Unicorn` gives you some magical template tags and HTML attributes to sprinkle in normal Django HTML templates. The backend code is a simple class that ultimately derives from `TemplateView`. Keep using the same Django HTML templates, template tags, filters, etc and the best-in-class Django ORM without learning another new framework of the week.

Do I need to build an entire API to use `Unicorn`*?*

Nope! `Django REST framework` is pretty magical on its own, and if you will need a mobile app or other use for a REST API, it’s a great set of abstractions to follow REST best practices. But, it can be challenging implementing a robust API even with `Django REST framework`. And I wouldn’t even attempt to build an API up from scratch unless it was extremely limited.

Do I need to install GraphQL to use `Unicorn`*?*

Nope! GraphQL is an awesome piece of technology for specific use-cases and solves some pain points around creating a RESTful API, but it is another thing to wrestle with.

Do I need to run an annoying separate node.js process or learn any tedious Webpack configuration incantations to use `Unicorn`*?*

Nope! `Unicorn` installs just like any normal Django package and is seamless to implement. There are a few “magic” attributes to sprinkle into a Django HTML template, but other than that it’s just like building a regular server-side application.

Does this replace Vue.js or React?

Nope! In some cases, you might need to actually build an SPA in which case `Unicorn` really isn’t that helpful. In that case you might have to invest the time to learn a more involved frontend framework. Read Using VueJS alongside Django for one approach, or check out other articles about this.

Isn't calling an AJAX endpoint on every input slow?

Not really! `Unicorn` is ideal for when an AJAX call would already be required (such as hitting an API for typeahead search or update data in a database). If that isn't required, the lazy and debounce modifiers can also be used to prevent an AJAX call on every change.

But, what about security?

`Unicorn` follows the best practices of Django and requires a CSRF token to be set on any page that has a component. This ensures that no nefarious AJAX POSTs can be executed. `Unicorn` also creates a unique component checksum with the Django secret key on every data change which also ensures that all updates are valid.

What browsers does `Unicorn` *support?*

`Unicorn` mostly targets modern browsers, but any PRs to help support legacy browsers would be appreciated.

How to make sure that the new JavaScript is served when a new version of `Unicorn` *is released?*

`Unicorn` works great using whitenoise to serve static assets with a filename based on a hash of the file. CompressedManifestStaticFilesStorage works great for this purpose and is used by django-unicorn.com for this very purpose. Example code can be found at https://github.com/adamghill/django-unicorn.com/.

What is the difference between `Unicorn` *and lighter front-end frameworks like* `htmx` *or* `alpine.js`*?*

htmx and alpine.js are great libraries to provide interactivity to your HTML. Both of those libraries are generalized front-end framework that you could use with any server-side framework (or just regular HTML). They are both well-supported, battle-tested, and answers to how they work are probably Google-able (or on Stackoverflow).

`Unicorn` isn't in the same league as either `htmx` or `alpine.js`. But, the benefit of `Unicorn` is that it is tightly integrated with Django and it should "feel" like an extension of the core Django experience. For example:

- redirecting from an action uses the Django `redirect` shortcut
- validation uses Django forms
- Django Models are tightly integrated
- Django messages "just work" the way you would expect them to
- you won't have to create extra URLs/views for AJAX calls to send back HTML because `Unicorn` handles all of that for you

Changelog

0.54.0

- Coerce type annotated arguments in an action method to the specified type (#571).
- Fix: Dictionary fields would sometimes create checksum errors (#572).

0.53.0

- Support passing arguments into a component (#560).
- Fix the handling of `None` for select elements. (#563).
- Better handling of `AuthenticationForm` when used in `Component.form_class` (#552) by lassebomh.

v0.52.0

- Use `CSRF_COOKIE_NAME` Django setting (#545) by frnidito.
- Asterisk wildcard support for targeting loading (#543) by regoawt.

v0.51.0

- Fix: remove use of `ByteString` (#534) by hauntsaninja.
- Fix: Update `loading` on elements other than the current action element ([#512]https://github.com/adamghill/django-unicorn/pull/512) by bazubii).
- Add new logo and doc changes (#518) by dancaron.
- Fix: Nested children caching issues (#511) by bazubii).
- Fix: Negating a variable for `poll.disable` would not work correctly in some instances.

v0.50.0

- Support more than 1 level of nested children (#476 by bazubii).

All changes since 0.49.2.

v0.49.2

- Fix: Calling methods with a model typehint would fail after being called multiple times (#476 by stat1c-void).

All changes since 0.49.1.

v0.49.1

- Fix: Missing `pp` import in Python 3.7.

All changes since 0.49.0.

v0.49.0

- Fix: Handle inherited (i.e. subclassed) models #459.
- Fix: Support abbreviated `u:view` (#464 by nerdoc).
- Add version and build date to minified JavaScript for easier debugging.

All changes since 0.48.0.

v0.48.0

- Reload JavaScript script elements when a template is re-rendered. Currently only enabled with the `RELOAD_SCRIPT_ELEMENTS` setting.

All changes since 0.47.0.

v0.47.0

- Fix: Include stacktrace for `AttributeError` errors.
- Fix: Only call `updated_` and `updating_` component functions once.

All changes since 0.46.0.

v0.46.0

- Support for loading nested components from a route that uses a class-based view.
- Better support for children components.

All changes since 0.45.1.

v0.45.1

- Fix: Handle JavaScript error that sometimes happens with nested components. 237 by clangley

All changes since 0.45.0.

v0.45.0

- Add ability to render initial data JavaScript inside the rendered component with `SCRIPT_LOCATION` setting

All changes since 0.44.1.

v0.44.1

- Fix: Some types of type annotations on a component method would cause an error when it was called #392 by nerdoc.
- Add `component_id`, `component_name`, `component_key` to the `unicorn` dictionary in the template context #389 by nerdoc.

All changes since 0.44.0.

v0.44.0

- Add support for raising a `ValidationError` from component methods.

All changes since 0.43.1.

v0.43.1

- Fix: direct views were not caching the component correctly.

All changes since 0.43.0.

v0.43.0

- Defer displaying `messages` when an action method returns a redirect.
- Prevent morphing or other changes when redirecting.

All changes since 0.42.1.

v0.42.1

- Fix: dictionaries in a component would generate incorrect checksums and trigger a `Checksum does not match` error
- Remove some serializations that was happening unnecessarily on every render.
- Add Python 3.10 and Django 4.0 to test matrix.

All changes since 0.42.0.

v0.42.0

- Remove all blank spaces from JSON responses.
- Optional support for minifying response HTML with `htmlmin`.
- Log warning message if the component HTML does not appear to be well-formed (i.e. an element does not have an ending tag). #342 by liamlawless35

Breaking changes

- Bump supported Python to >=3.7.

All changes since 0.41.2.

v0.41.2

- Fix: Handle excluding a field's attribute when the field is `None`.

All changes since 0.41.1.

v0.41.1

- Fix: Handle component classes with a `bool` class attribute and a `form_class` with a `BooleanField`. Reported by zurtri

All changes since 0.41.0.

v0.41.0

- Support using a context variable for a component name. #314 by robwa

All changes since 0.40.0.

v0.40.0

- Add direct view so that components can be added directly to urls without being required to be included in a regular Django template.
- Add capability for `startunicorn` to create components in sub-folders. (#299)[https://github.com/adamghill/django-unicorn/issues/299]

All changes since 0.39.1.

v0.39.1

- Prefer `prefetch_related` to reduce database calls for many-to-many fields.

All changes since 0.39.0.

v0.39.0

- Explicit error messages when an invalid component field is excluded
- Better support for serializing many-to-many fields which have been prefetched to reduce the number of database calls
- Support excluding many-to-many fields with `javascript_exclude`

All changes since 0.38.1.

v0.38.1

- Fix: Allow components to be `pickled` so they can be cached.

All changes since 0.38.0.

v0.38.0

- Include request context in component templates.

All changes since 0.37.2.

v0.37.2

- Fix: nested field attributes for `javascript_exclude`.

All changes since 0.37.1.

v0.37.1

- Support nested field attributes for `javascript_exclude`.

All changes since 0.37.0.

v0.37.0

- Revert loading and dirty elements when the server returns a 304 (not modified) or a 500 error.

All changes since 0.36.1.

v0.36.1

- More verbose error messages when components can't be loaded (nerdoc).
- More complete handling to prevent XSS attacks.

All changes since 0.36.0.

v0.36.0

- Security fix: for CVE-2021-42053 to prevent XSS attacks (reported by Jeffallan).

** Breaking changes **

- responses will be HTML encoded going forward (to explicitly opt-in to previous behavior use safe)

All changes since 0.35.3.

v0.35.3

- Fix: Handle when there are multiple apps sub-directories 273 by apoorvaeternity.

All changes since 0.35.2.

v0.35.2

- Fix: Make sure `visible:elements` trigger as expected in more cases.
- Prevent the visible element from continuing to trigger if the visibility element method returns `False`.

All changes since 0.35.0.

v0.35.0

- Trigger an `input` or `blur` event for a model element from JavaScript.
- Visibility event with `unicorn:visible` attribute.

Breaking changes

- `db_model` Python decorator, `unicorn:db`, `unicorn:field`, `unicorn:pk` template attributes are removed.

All changes since 0.34.0.

v0.34.0

- Initial prototype for component template lifecycle events.
- Fix: elements after a child component would not get initialized #262 by joshiggins.
- Fix: cache would fail in some instances 258.

All changes since 0.33.0.

v0.33.0

- Fix: Allow comments, blank lines, or text at the top of component templates before the root element.

All changes since 0.32.0.

v0.32.0

- Add debounce support to actions.

All changes since 0.31.0.

v0.31.0

- Move JavaScript static assets into `unicorn` sub-folder
- Determine correct path for installed app passed to `startunicorn` management command
- Call `startapp` management command if app is not already installed

All changes since 0.30.0.

v0.30.0

- Look in all `INSTALLED_APPS` for components instead of only in a `unicorn` app 210
- Support `settings.APPS_DIR` which is the default for `django-cookiecutter` instead of just `settings.BASE_DIR` 214

** Breaking changes **

- Require an application name when running the `startunicorn` management command for where the component should be created

All changes since 0.29.0.

v0.29.0

- Sanitize initial JSON to prevent XSS

All changes since 0.28.0.

v0.28.0

- Re-fire poll method when tab/window comes back into focus after losing visibility (https://github.com/adamghill/django-unicorn/pull/202 by frbor)

All changes since 0.27.2.

v0.27.2

- Fix bug with relationship fields on a Django model

All changes since 0.27.1.

v0.27.1

- Fix some issues with many-to-many fields on a Django model

All changes since 0.27.0.

v0.27.0

- Many-to-many fields on a Django model are now supported
- Multiple partial targets

All changes since 0.26.0.

v0.26.0

- Completely redesigned and much improved support for Django models and QuerySets.
- Fix the `startunicorn` command and add some ascii art.

All changes since 0.25.0.

v0.25.0

- Support calling functions in JavaScript modules.
- Fix: use `unicorn:db` without a `unicorn:model` in the same element.

All changes since 0.24.0.

v0.24.0

- Support custom CSRF headers set with CSRF_HEADER_NAME setting.

All changes since 0.23.0.

v0.23.0

- Performance enhancement that returns a 304 HTTP status code when an action happens, but the content doesn't change.
- Add `unicorn:ignore` attribute to prevent an element from being morphed (useful when using `Unicorn` with libraries like `Select2` that change the DOM).
- Add support for passing arguments to `Unicorn.call`.
- Bug fix when attempting to cache component views that utilize the `db_model` decorator.

All changes since 0.22.0.

v0.22.0

- Use Django cache for storing component state when available
- Add support for Django 2.2.x

All changes since 0.21.2.

v0.21.2

- Add backported `dataclasses` for Python 3.6. (@frbor)

All changes since 0.21.0.

v0.21.0

- Bug fix: Prevent disabled polls from firing at all.
- Support `Decimal` field type.
- Support `dataclass` field type.
- Use type hints to cast fields to primitive Python types if possible.

All changes since 0.20.0.

v0.20.0

- Add ability to exclude component view properties from JavaScript to reduce the amount of data initially rendered to the page with `javascript_exclude`.
- Add `complete`, `rendered`, `parent_rendered` component hooks.
- Call JavaScript functions from a component view's method.

All changes since 0.19.0.

v0.19.0

- Re-implemented how action method parsing is done to remove all edge cases when passing arguments to component view methods. (@frbor).
- Add support for passing kwargs to component view methods.

All changes since 0.18.1.

v0.18.1

- Fix regression where component kwargs were getting lost (#140, #141)
- Fix <code>startunicorn</code> management command (#142)

All changes since 0.18.0.

v0.18.0

- Only send updated data back in the response to reduce network latency.
- Experimental support for queuing up requests to alleviate race conditions when functions take a long time to process.

- Bug fix: prevent race condition where an instantiated component class would be inadvertently re-used for component views that are slow to render
- Bug fix: use the correct component name to call a component method from “outside” the component.
- Deprecated: `DJANGO_UNICORN` setting has been renamed to `UNICORN`.

All changes since 0.17.2.

v0.17.2

- Don’t send the parent context in the response for child components that specify a partial update.
- Add support for element models to specify a partial update.
- Add support for polls to specify a partial update.
- Handle `date`, `time`, `timespan` when passed as arguments from JavaScript.
- Render child component template’s JavaScript initialization with the parent’s as opposed to inserting a new script tag after the child component is rendered.
- Bug fix: prevent an error when rendering a Django model with a date-related field, but a string value.

All changes since 0.17.1.

v0.17.1

- Remove stray print statement.
- Fix bug where child components would sometimes lose their action events.

All changes since 0.17.0.

v0.17.0

- Target DOM changes from an action to only a portion of the DOM with partial updates.

All changes since 0.16.1.

v0.16.1

- Remove debounce from action methods to reduce any perceived lag.

All changes since 0.16.0.

v0.16.0

- Dirty states for when there is a change that hasn’t been synced yet.
- Add support for setting multiple classes for loading states.
- Attempt to handle when the component gets out of sync with an invalid checksum error.
- Performance tweaks when there isn’t a change to a model or dbModel with lazy or defer modifiers.

All changes since 0.15.1.

v0.15.1

- Fix bug where a component name has a dash in its name

All changes since 0.15.1.

v0.15.0

- Add support for child components
- Add discard action modifier
- Add support for referring to components in a folder structure
- Remove restriction that component templates must start with a div
- Remove restriction that component root can't also have `unicorn:model` or `unicorn:action`

All changes since 0.15.0.

v0.14.1

- Prevent the currently focused model element from updating after the AJAX request finishes (#100).

All changes since 0.14.0.

v0.14.0

- Disable poll with a component field
- Dynamically change polling options with PollUpdate
- Basic support for `pydantic` models

All changes since 0.13.0.

v0.13.0

- Component key to allow disambiguation of components of the same name
- `$returnValue` special argument
- Get the last action method's return value

All changes since 0.12.0.

v0.12.0

- Redirect from action method in component

All changes since 0.11.2.

v0.11.2

- Fix encoding issue with default component template on Windows (#91)
- Fix circular import when creating the component (#92)

All changes since 0.11.0.

v0.11.0

- `$toggle` special method.
- Support nested properties when using the set shortcut.
- Fix action string arguments that would get spaces removed inadvertently.

Breaking changes

- All existing special methods now start with a `$` to signify they are magical. Therefore, `refresh` is now `$refresh`, `reset` is now `$reset`, and `validate` is now `$validate`.

All changes since 0.10.1.

v0.10.1

- Use LRU cache for constructed components to prevent ever-expanding memory.
- Loosen `beautifulsoup4` version requirement.
- Fix bug to handle floats so that they don't lose precision when serialized to JSON.
- Fix bug to handle related models (ForeignKeys, OneToOne, etc) fields in Django models.

All changes since 0.10.0.

v0.10.0

- Add support for passing kwargs into the component on the template
- Provide access to the current request in the component's methods

All changes since 0.9.4.

v0.9.4

- Fix: Prevent Django `CharField` form field from stripping whitespaces when used for validation.
- Fix: Handle edge case that would generate a null exception.
- Fix: Only change loading state when an action method gets called, not on every event fire.

All changes since 0.9.1.

v0.9.3

- Handle child elements triggering an event which should be handled by a parent unicorn element.

All changes since 0.9.1.

v0.9.1

- Fix: certain actions weren't triggering model values to get set correctly

All changes since 0.9.0.

v0.9.0

- Loading states for improved UX.
- `$event` special argument for `actions`.
- u unicorn attribute.
- APPS setting for determing where to look for components.
- Add support for parent elements for non-db models.
- Fix: Handle if `Meta` doesn't exist for db models.

All changes since 0.8.0.

v0.8.0

- Add much more elaborate support for dealing with Django models.

All changes since 0.7.1.

v0.7.1

- Fix bug where multiple actions would trigger multiple payloads.
- Handle lazy models that are children of an action model better.

All changes since 0.7.0.

v0.7.0

- Parse action method arguments as basic Python objects
- Stop and prevent modifiers on actions
- Defer modifier on model
- Support for multiple actions on the same element
- Django setting for whether the JavaScript is minified

Breaking changes

- Remove unused `unicorn_styles` template tag
- Use dash for poll timing instead of dot

All changes since 0.6.5.

v0.6.5

- Attempt to get the CSRF token from the cookie first before looking at the CSRF token.

All changes since 0.6.4.

v0.6.4

- Fix bug where lazy models weren't sending values before an action was called
- Add `is_valid` method to component to more easily check if a component has validation errors.
- Better error message if the CSRF token is not available.

All changes since 0.6.3.

v0.6.3

- Fix bug where model elements weren't getting updated values when an action was being called during the same component update.
- Fix bug where some action event listeners were duplicated.

All changes since 0.6.2.

v0.6.2

- More robust fix for de-duping multiple actions.
- Fix bug where conditionally added actions didn't get an event listener.

All changes since 0.6.1.

v0.6.1

- Fix model sync getting lost when there is an action (issue 39).
- Small fix for validations.

All changes since 0.6.0.

v0.6.0

- Realtime validation of a Unicorn model.
- Polling for component updates.
- More component hooks

All changes since 0.5.0.

v0.5.0

- Call component method from JavaScript.
- Support classes, dictionaries, Django Models, (read-only) Django QuerySets properties on a component.
- Debounce modifier to change how fast changes are sent to the backend from `unicorn:model`.
- Lazy modifier to listen for `blur` instead of `input` on `unicorn:model`.
- Better support for `textarea` HTML element.

All changes since 0.4.0.

v0.4.0

- Set shortcut for setting properties.
- Listen for any valid event, not just `click`.
- Better handling for model updates when element ids aren't unique.

All changes since 0.3.0.

v0.3.0

- Add mount hook.
- Add reset action.
- Remove lag when typing fast in a text input and overall improved performance.
- Better error handling for exceptional cases.

All changes since 0.2.3.

v0.2.3

- Fix for creating default folders when running `startunicorn`.

All changes since 0.2.2.

v0.2.2

- Set default `template_name` if it's missing in component.

All changes since 0.2.1.

v0.2.1

- Fix `startunicorn` Django management command.

All changes since 0.2.0.

v0.2.0

- Switch from `Component` class to `UnicornView` to follow the conventions of class-based views.
- Investigate using class-based view instead of the custom Component class

All changes since 0.1.1.

v0.1.1

- Fix package readme and repository link.

All changes since 0.1.0.

v0.1.0

- Initial version with basic functionality.

Troubleshooting

Disallowed MIME type error on Windows

Apparently Windows system-wide MIME type configuration sometimes won't load up JavaScript modules in certain browsers. The errors would be something like `Loading module from “http://127.0.0.1:8000/static/js/unicorn.js” was blocked because of a disallowed MIME type (“text/plain”)` or `Failed to load module script: The server responded with a non-JavaScript MIME type of "text/plain".`

One suggested solution is to add the following to the bottom of the settings file:

```
# settings.py

if DEBUG:
    import mimetypes
    mimetypes.add_type("application/javascript", ".js", True)
```

See this Windows MIME type detection pitfalls article, this StackOverflow answer, or issue #201 for more details.

Architecture

`Unicorn` is made up of multiple pieces which are all integrated tightly together. The following is a summary of how some of it all fits together, although it skips over a lot of the complexity and advanced functionality. However, for all of the details the code is available at https://github.com/adamghill/django-unicorn/.

Template tags

Starting with the integration with a normal Django template, there are the `unicorn_scripts` and `unicorn` template tags. `unicorn_scripts` renders out the entire JavaScript library and initializes the global `Unicorn` object. The `unicorn` template tag provides the ability to add the component wherever it is needed on the page. Based on the name passed into the `unicorn` template tag, conventions are used to find the correct component view and component template (e.g. if “hello-world” is passed into the template tag, a class of `hello_world.HelloWorldView` and a template named `hello-world.html` will be searched for).

Once the component view and template are found, a serialized version of all of the public attributes of the component view is generated into a JSON object for the page, and the template is rendered with a context of those same public attributes.

JavaScript initialization

After the template is rendered, the JavaScript library parses the HTML for DOM elements that start with `unicorn:` or `u:` and creates a list of attributes that end with `:model`, `:poll`, or other specific `Unicorn` functionality. For attributes that are left, the assumption is that they are an event type (e.g. `unicorn:click`).

For anything that is a model, the JavaScript sets the value for the element based on the serialized data of the publicly available attributes from the component view. Event listeners are attached for all event types. Then, other custom functionality is setup (e.g. polling).

Models

For all inputs which have a `model` attribute, an event listener is attached (either `change` or `blur` depending on if the `lazy` modifier is used). The `defer` modifier will store the action to be bundled with an action event that might happen later.

Once a model event is fired it is sent over the wire to the defined AJAX endpoint with a specific JSON structure which tells `Unicorn` what the updated data from the input should be. The component class is re-instantiated and the data is updated from the front-end, then re-rendered and the HTML is returned in the response.

Actions

Actions follow a similar path as the models above, however there is a different JSON stucture. Also, the method, arguments, and kwargs that are passed from the front-end get parsed with a mix of `ast.parse` and `ast.literal_eval` to convert the strings into the appropriate Python types (i.e. change the string “1” to the integer `1`). After the component is re-initialized, the method is called with the passed-in arguments and kwargs. Once all of the actions have been called, the component view is re-rendered and the HTML is returned in the response.

HTML Diff

After the AJAX endpoint returns its response, the newly rendered DOM is merged into the old DOM with `morphdom` and input values are set again based on the new data in the AJAX response.

Contributor Covenant Code of Conduct

Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, caste, color, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at conduct@adamghill.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

Want to add some component-based magic to your front-end, but don't need the overhead of a complete JavaScript front-end framework? `Unicorn` revolutionizes the way your users interact with your Django app! With `Unicorn`, you can create stunningly spiffy pages without ever leaving Python or your beloved Django codebase.

`Unicorn` is a reactive component framework that enhances your Django views by seamlessly making AJAX calls in the background and dynamically updates the HTML DOM. It's like magic, but better! `Unicorn` is leading the charge in bringing a component-based developer experience to Django. Join the `Unicorn` community today and unlock the power of reactivity!

Here are a few reasons to consider `Unicorn`.

Reactive Components: With `Unicorn`, you can create reactive components that dynamically update the HTML DOM without the need for complex JavaScript. This makes it easier to build interactive web pages and enhances the user experience.

Seamless Integration: `Unicorn` progressively enhances your Django views. This means you can seamlessly integrate Unicorn into your existing Django codebase without disrupting your current workflow.

Component-Based Design: `Unicorn` brings the benefits of component-based design to the Python ecosystem, making it easier to build complex applications and enabling more efficient development.

Improved Performance: By using AJAX calls to update the DOM, `Unicorn` reduces the need for full page reloads, which can result in improved performance and faster load times.

Familiarity: With `Unicorn`, you don't need to learn a complicated front-end frameworks to create fancy interactive components. Instead, you can use the familiar Django syntax and templates to build your front-end components.

Related projects

`Unicorn` stands on the shoulders of giants, in particular morphdom which is integral for merging DOM changes.

Inspirational projects in other languages

- Livewire, a full-stack framework for the PHP web framework, Laravel.
- LiveView, a library for the Elixir web framework, Phoenix, that uses websockets.
- StimulusReflex, a library for the Ruby web framework, Ruby on Rails, that uses websockets.
- Hotwire, “is an alternative approach to building modern web applications without using much JavaScript by sending HTML instead of JSON over the wire”. Uses AJAX, but can optionally use websockets.

Full-stack framework Python packages

- Reactor, a port of Elixir’s `LiveView` to Django. Especially interesting for more complicated use-cases like chat rooms, keeping multiple browsers in sync, etc. Uses Django channels and websockets to work its magic.
- Flask-Meld, a port of `Unicorn` to Flask. Uses websockets.
- Sockpuppet, a port of Ruby on Rail’s `StimulusReflex`. Requires Django channels and websockets.
- Django inertia.js adapter allows Django to use inertia.js to build an SPA without building an API.
- Hotwire for Django contains a few different repositiories to integrate Hotwire with Django.
- Lona is a web application framework, designed to write responsive web apps in full Python.
- ReactPy is a port of ReactJS to Python. Fully compatible with all ReactJS components.
- django-async-include load HTML via AJAX.

Django component packages

- django-components, which lets you create “template components”, that contains both the template, the Javascript and the CSS needed to generate the front end code you need for a modern app.
- django-component, which provides declarative and composable components for Django, inspired by JavaScript frameworks.
- django-page-components, a minimalistic framework for creating page components and using them in your Django views and templates.
- slippers, helps build reusable components in Django without writing a single line of Python.
- django_slots allows multiline strings to be captured and passed to template tags.

Django packages to integrate lightweight frontend frameworks

- django-htmx which has extensions for using Django with htmx.