Unicorn

version

Adam Hill

January 17, 2021

Contents

Unicorn

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` array in the Django settings file (normally `settings.py`)

```
# settings.py
INSTALLED_APPS = (
    # other apps
    "django_unicorn",
)
```

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 %}` and `{% csrf_token %}` into a Django HTML template

```
<!-- 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 class which provides the backend code for the template.

Create a component

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

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

Warning

If this is the first component you create, you will also need to add "unicorn", to INSTALLED_APPS in your Django settings file (normally settings.py) to make sure that Django can find the created component templates.

Also, make sure that there is a {% csrf_token %} in your HTML somewhere to prevent cross-site scripting attacks while using Unicorn.

Note

Change which apps Unicorn looks in for components with the APPS setting.

Add {% unicorn 'hello-world' %} into the template where you want to load the new component.

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' %}.

Component arguments

kwargs can be passed into the unicorn templatetag from the template. The kwargs will be available in the component __init__ method.

Warning

When overriding __init__ calling super().__init__(**kwargs) is required for the component to initialize properly.

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

class HelloWorldView(UnicornView):
    name = "World"

    def __init__(self, *args, **kwargs):
        super().__init__(**kwargs)  # calling super is required
        self.name = kwargs.get("name")
```

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

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

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

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

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

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>
```

`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).

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`, `Django Model`, `Django QuerySet`, or `custom classes`.

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`.

```python
# 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.

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

```
class Author(UnicornField):
    def __init__(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.

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.

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.

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]
```

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.

```
<!-- 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 Django workflows. There are multiple ways to integrate a Django model with a component. They all work a little differently and which option you choose to use depends on the situation.

DbModel

One way to use Django models is by utilizing the `DbModel` class that provides a way from the front-end to refer to a Django model. The value of the `unicorn:db` attribute refers to the first argument when constructing a `DbModel` and `unicorn:field` is used for the model's field that should be bound to the input.

Another benefit of `DbModel` is that it enables the use of the `$model` special action variable.

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

```
# db_model.py
from django_unicorn.components import UnicornView
from django_unicorn.db import DbModel
from books.models import Book

class DbModelView(UnicornView):
    class Meta:
        db_models = [DbModel("book", Book)]

    def save(self):
        print("A new book will be created automatically")
        pass
```

`unicorn:db` can also live in a parent element and surround a group of `unicorn:field` inputs.

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

`unicorn:pk` can be used so that an existing model is updated instead of a new model is created. More information about `unicorn:pk` is in queryset.

```
<!-- db-model.html -->
<div>
  <div unicorn:db="book">
    <div unicorn:pk="1">
      <div>
        <input unicorn:field.defer="title" type="text" id="title" />
        {{ book.title }}
      </div>
      <div>
```

```
        <input unicorn:field.defer="description" type="text" id="description" />
        {{ book.description }}
      </div>
    </div>
  </div>
  <button unicorn:click="save">Save</button>
</div>
```

Class Model

Django models can be initialized similar to how basic Python objects (i.e. strings, integers, dictionaries) can be set on a component.

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

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

class ClassModelView(UnicornView):
    book = Book()

    def save(self):
        self.book.save()
```

Instance Model

Django models can be initialized in the component's `__init__` method similar to how a "normal" class would initialize an instance variable.

!DANGER!

`super().__init__(**kwargs)` **has** to be called at the end of the overriden `__init__` method.

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

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

class InstanceModelView(UnicornView):
    def __init__(self, **kwargs):
        self.book = Book()

        # super() has to be called at the end
        super().__init__(**kwargs)
```

```
    def save(self):
        self.book.save()
```

Queryset

Binding models to a Django `Queryset` is done by setting an `unicorn:pk` attribute with the `model`'s `pk` (normally an integer in an `id` field, but could be a custom `primary_key`).

Warning

A blank value for an `unicorn:pk` attribute signals to `Unicorn` to create a new instance of the underlying Django `model` of the queryset.

```
<!-- queryset.html -->
<div>
  <div unicorn:model="books">
    <div unicorn:pk="">
      <!-- A blank pk will create a new model when it is saved -->
      <div>
        <input unicorn:field.defer="title" type="text" id="title" />
      </div>
      <div>
        <input unicorn:field.defer="description" type="text" id="description" />
      </div>
    </div>

    {% for book in books %}
    <div unicorn:pk="{{ book.pk }}">
      <!-- Using the model's pk will save the model -->
      <div>
        <input unicorn:field.defer="title" type="text" id="title" />
        {{ book.title }}
      </div>
      <div>
        <input unicorn:field.defer="description" type="text" id="description" />
        {{ book.description }}
      </div>
    </div>
    {% endfor %}

  </div>
  <button unicorn:click="save">Save</button>
</div>
```

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

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

    def hydrate(self):
        # Using `hydrate` is the best way to make sure that QuerySets
        # are re-queried every time the component is loaded
        self.books = Book.objects.all().order_by("-id")[:5]

    def save(self):
        pass
```

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

Arguments can be most basic Python types, including `string`, `int`, `float`, `list`, `tuple`, `dictionary`, `set`, `datetime`, and `UUID4`.

```
<!-- 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({'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>
  <button unicorn:click="update(2020-09-12T01:01:01)">Pass datetime</button>
  <button unicorn:click="update(90144cb9-fc47-476d-b124-d543b0cff091)">
    Pass UUID
  </button>
</div>
```

> ***Note***
>
> Strings will be converted to `datetime` if they are successfully parsed by Django's `parse_datetime` method.

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
```

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>
```

$model

Sends the current `db_model` to an action.

> ***Note***
>
> `$model` requires `db_models` to be defined in the component's `Meta` class. The component method must also be decorated with `django_unicorn.decorators.db_model` and must have at least one argument (which will be converted into the specified Django model from the frontend).

```
# model.py
from django_unicorn.components import UnicornView
from django_unicorn.db import DbModel
from django_unicorn.decorators import db_model
from .models import Book

class ModelView(UnicornView):
    books = Book.models.all()

    @db_model
    def delete(self, book):
        book.delete()

    class Meta:
        db_models = [DbModel("book", Book)]
```

```
<!-- model.html -->
<div>
```

```
    <div unicorn:db="book">
        {% for book in books %}
        <div unicorn:pk="{{ book.pk }}">
            <input type="text" unicorn:change="delete($model)">Delete the current book</inpu
        </div>
        {% endfor %}
    </div>
</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()`. It can be called from anywhere on the page.

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

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

Validation

`Unicorn` uses Django `forms` infrastructure for all validation. This means that a form could be re-used between any other Django views and a `Unicorn`.

Using the Django `forms` system provides a way to serialize/deserialize certain classes (for example, `datetime` and `uuid`) and a way to validate properties of a class.

Note

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

```
# book.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 = ""
```

```
<!-- book.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.

Showing validation errors

As the form is filled out the appropriate inputs will be validated. 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 should provide 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>
```

Validate the entire component

The magic action method `$validate` can be used to validate the whole component by the front-end.

```
<!-- 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.

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

```
# 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"
```

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}")
```

```html
<!-- 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

Because `Unicorn` requires an AJAX request for any component updates, 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 specifc action by targetting 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>
```

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 to the element that is triggering the action.

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

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

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
```

```python
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.

```python
# 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>
```

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>
```

```python
# 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")
```

Advanced

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

request

The current `request` is available on `self` in the component's methods.

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

class HelloWorldView(UnicornView):
    def __init__(self, *args, **kwargs):
        super.__init__(**kwargs)
        print("Initial request that rendered the component", self.request)

    def test(self):
        print("callMethod request to re-render 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

__init__()

Gets called when the component gets constructed for the very first time. Note that constructed components get cached to reduce the amount of time discovering and instantiating them, so `__init__` only gets called the very first time the component gets rendered.

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

class HelloWorldView(UnicornView):
    name = "original"

    def __init__(self, *args, **kwargs):
        super().__init__(**kwargs)
        self.name = "initialized"
```

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.

Meta

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

exclude

By default, all properties of the component are included in the payload when hydrating the Django template. One way to prevent internal-only data from getting POSTed needlessly is to prefix the property name with _ to indicate it should stay private.

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

class HelloStateView(UnicornView):
    state = ""

    _all_states = (
        "Alabama",
        "Alaska",
        "Arizona",
        "Arkansas",
        ...
        "Wisconsin",
        "Wyoming",
    )
```

Another way to prevent that data from going across the wire on every interaction is to add it to the `Meta` class's `exclude` tuple.

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

class HelloStateView(UnicornView):
    state = ""

    all_states = (
        "Alabama",
```

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

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

Settings

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

```
# settings.py
DJANGO_UNICORN = {
    "MINIMIZE": True,
    "APPS": ["unicorn"]
}
```

MINIMIZE

Provides a way to control if the minimized version of `unicorn.min.js` is used. Defaults to `!DEBUG`.

APPS

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

CLI

`Unicorn` provides a Django management command to create new components.

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

The command will 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/
        hello_world.py
    templates/
        unicorn/
            hello-world.html
```

Changelog

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

- `$model` special argument and decorator.
- `$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 to minimize the JavaScript

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.

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

Nope! GraphQL looks like an awesome technology for specific use-cases and solves some pain points around creating a RESTful API. But, it is another peiece of technology 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 the project would appreciate any PRs to help support legacy browsers.

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

`Unicorn` works great with the whitenoise ability 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 into `Unicorn` (especially with the $model special argument)
- 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

`Unicorn` is a reactive component framework that progressively enhances a normal Django view, makes AJAX calls in the background, and dynamically updates the DOM. It seamlessly extends Django past its server-side framework roots without giving up all of its niceties or re-building your website.

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 also 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 or its separate libraries into Django.

Django packages to integrate lightweight frontend frameworks

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

Django component packages

- django-components, 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.