Form Assets (the Media class)

Rendering an attractive and easy-to-use web form requires more than just HTML - it also requires CSS stylesheets, and if you want to use fancy widgets, you may also need to include some JavaScript on each page. The exact combination of CSS and JavaScript that is required for any given page will depend upon the widgets that are in use on that page.

This is where asset definitions come in. Django allows you to associate different files – like stylesheets and scripts – with the forms and widgets that require those assets. For example, if you want to use a calendar to render DateFields, you can define a custom Calendar widget. This widget can then be associated with the CSS and JavaScript that is required to render the calendar. When the Calendar widget is used on a form, Django is able to identify the CSS and JavaScript files that are required, and provide the list of file names in a form suitable for inclusion on your web page.

Assets and Django Admin: The Django Admin application defines a number of customized widgets for calendars, filtered selections, and so on. These widgets define asset requirements, and the Django Admin uses the custom widgets in place of the Django defaults. The Admin templates will only include those files that are required to render the widgets on any given page.

If you like the widgets that the Django Admin application uses, feel free to use them in your own application! They're all stored in `django.contrib.admin.widgets`.

Which JavaScript toolkit?: Many JavaScript toolkits exist, and many of them include widgets (such as calendar widgets) that can be used to enhance your application. Django has deliberately avoided blessing any one JavaScript toolkit. Each toolkit has its own relative strengths and weaknesses - use whichever toolkit suits your requirements. Django is able to integrate with any JavaScript toolkit.

Assets as a static definition

The easiest way to define assets is as a static definition. Using this method, the declaration is an inner `Media` class. The properties of the inner class define the requirements.

Here's an example:

```
from django import forms


class CalendarWidget(forms.TextInput):
    class Media:
        css = {
            "all": ["pretty.css"],
        }
        js = ["animations.js", "actions.js"]
```

This code defines a `CalendarWidget`, which will be based on `TextInput`. Every time the CalendarWidget is used on a form, that form will be directed to include the CSS file `pretty.css`, and the JavaScript files `animations.js` and `actions.js`.

This static definition is converted at runtime into a widget property named `media`. The list of assets for a `CalendarWidget` instance can be retrieved through this property:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>
```

Here's a list of all possible `Media` options. There are no required options.

```
css
```

A dictionary describing the CSS files required for various forms of output media.

The values in the dictionary should be a tuple/list of file names. See the section on paths for details of how to specify paths to these files.

The keys in the dictionary are the output media types. These are the same types accepted by CSS files in media declarations: 'all', 'aural', 'braille', 'embossed', 'handheld', 'print', 'projection', 'screen', 'tty' and 'tv'. If you need to have different stylesheets for different media types, provide a list of CSS files for each output medium. The following example would provide two CSS options – one for the screen, and one for print:

```
class Media:
    css = {
        "screen": ["pretty.css"],
        "print": ["newspaper.css"],
    }
```

If a group of CSS files are appropriate for multiple output media types, the dictionary key can be a comma separated list of output media types. In the following example, TV's and projectors will have the same media requirements:

```
class Media:
    css = {
        "screen": ["pretty.css"],
        "tv,projector": ["lo_res.css"],
        "print": ["newspaper.css"],
    }
```

If this last CSS definition were to be rendered, it would become the following HTML:

```
<link href="https://static.example.com/pretty.css" media="screen" rel="stylesheet">
<link href="https://static.example.com/lo_res.css" media="tv,projector" rel="stylesheet">
<link href="https://static.example.com/newspaper.css" media="print" rel="stylesheet">
```

`js`

A tuple describing the required JavaScript files. See the section on paths for details of how to specify paths to these files.

`extend`

A boolean defining inheritance behavior for `Media` declarations.

By default, any object using a static `Media` definition will inherit all the assets associated with the parent widget. This occurs regardless of how the parent defines its own requirements. For example, if we were to extend our basic Calendar widget from the example above:

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         css = {
...             "all": ["fancy.css"],
...         }
...         js = ["whizbang.js"]
...

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<link href="https://static.example.com/fancy.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>
<script src="https://static.example.com/whizbang.js"></script>
```

The FancyCalendar widget inherits all the assets from its parent widget. If you don't want `Media` to be inherited in this way, add an `extend=False` declaration to the `Media` declaration:

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         extend = False
...         css = {
...             "all": ["fancy.css"],
...         }
...         js = ["whizbang.js"]
...

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="https://static.example.com/fancy.css" media="all" rel="stylesheet">
<script src="https://static.example.com/whizbang.js"></script>
```

If you require even more control over inheritance, define your assets using a dynamic property. Dynamic properties give you complete control over which files are inherited, and which are not.

`Media` as a dynamic property

If you need to perform some more sophisticated manipulation of asset requirements, you can define the `media` property directly. This is done by defining a widget property that returns an instance of `forms.Media`. The constructor for `forms.Media` accepts `css` and `js` keyword arguments in the same format as that used in a static media definition.

For example, the static definition for our Calendar Widget could also be defined in a dynamic fashion:

```
class CalendarWidget(forms.TextInput):
    @property
    def media(self):
        return forms.Media(
            css={"all": ["pretty.css"]}, js=["animations.js", "actions.js"]
        )
```

See the section on Media objects for more details on how to construct return values for dynamic `media` properties.

Paths in asset definitions

Paths as strings

String paths used to specify assets can be either relative or absolute. If a path starts with `/`, `http://` or `https://`, it will be interpreted as an absolute path, and left as-is. All other paths will be prepended with the value of the appropriate prefix. If the `django.contrib.staticfiles` app is installed, it will be used to serve assets.

Whether or not you use `django.contrib.staticfiles`, the `STATIC_URL` and `STATIC_ROOT` settings are required to render a complete web page.

To find the appropriate prefix to use, Django will check if the `STATIC_URL` setting is not `None` and automatically fall back to using `MEDIA_URL`. For example, if the `MEDIA_URL` for your site was `'https://uploads.example.com/'` and `STATIC_URL` was `None`:

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             "all": ["/css/pretty.css"],
...         }
...         js = ["animations.js", "https://othersite.com/actions.js"]
...

>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" media="all" rel="stylesheet">
<script src="https://uploads.example.com/animations.js"></script>
<script src="https://othersite.com/actions.js"></script>
```

But if `STATIC_URL` is `'https://static.example.com/'`:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://othersite.com/actions.js"></script>
```

Or if `staticfiles` is configured using the `ManifestStaticFilesStorage`:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.27e20196a850.js"></script>
<script src="https://othersite.com/actions.js"></script>
```

Paths as objects

Asset paths may also be given as hashable objects implementing an `__html__()` method. The `__html__()` method is typically added using the `html_safe()` decorator. The object is responsible for outputting the complete HTML `<script>` or `<link>` tag content:

```
>>> from django import forms
>>> from django.utils.html import html_safe
>>>
>>> @html_safe
... class JSPath:
...     def __str__(self):
...         return '<script src="https://example.org/asset.js" defer>'
...

>>> class SomeWidget(forms.TextInput):
...     class Media:
...         js = [JSPath()]
...
```

`Media` objects

When you interrogate the `media` attribute of a widget or form, the value that is returned is a `forms.Media` object. As we have already seen, the string representation of a `Media` object is the HTML required to include the relevant files in the `<head>` block of your HTML page.

However, `Media` objects have some other interesting properties.

Subsets of assets

If you only want files of a particular type, you can use the subscript operator to filter out a medium of interest. For example:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>

>>> print(w.media["css"])
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
```

When you use the subscript operator, the value that is returned is a new `Media` object – but one that only contains the media of interest.

Combining `Media` objects

`Media` objects can also be added together. When two `Media` objects are added, the resulting `Media` object contains the union of the assets specified by both:

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             "all": ["pretty.css"],
...         }
...         js = ["animations.js", "actions.js"]
...

>>> class OtherWidget(forms.TextInput):
...     class Media:
...         js = ["whizbang.js"]
...

>>> w1 = CalendarWidget()
>>> w2 = OtherWidget()
>>> print(w1.media + w2.media)
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>
<script src="https://static.example.com/whizbang.js"></script>
```

Order of assets

The order in which assets are inserted into the DOM is often important. For example, you may have a script that depends on jQuery. Therefore, combining `Media` objects attempts to preserve the relative order in which assets are defined in each `Media` class.

For example:

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         js = ["jQuery.js", "calendar.js", "noConflict.js"]
...
>>> class TimeWidget(forms.TextInput):
...     class Media:
...         js = ["jQuery.js", "time.js", "noConflict.js"]
...
>>> w1 = CalendarWidget()
>>> w2 = TimeWidget()
>>> print(w1.media + w2.media)
<script src="https://static.example.com/jQuery.js"></script>
<script src="https://static.example.com/calendar.js"></script>
<script src="https://static.example.com/time.js"></script>
<script src="https://static.example.com/noConflict.js"></script>
```

Combining `Media` objects with assets in a conflicting order results in a `MediaOrderConflictWarning`.

`Media` on Forms

Widgets aren't the only objects that can have `media` definitions – forms can also define `media`. The rules for `media` definitions on forms are the same as the rules for widgets: declarations can be static or dynamic; path and inheritance rules for those declarations are exactly the same.

Regardless of whether you define a `media` declaration, *all* Form objects have a `media` property. The default value for this property is the result of adding the `media` definitions for all widgets that are part of the form:

```
>>> from django import forms
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)
...

>>> f = ContactForm()
>>> f.media
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>
<script src="https://static.example.com/whizbang.js"></script>
```

If you want to associate additional assets with a form – for example, CSS for form layout – add a `Media` declaration to the form:

```
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)
...     class Media:
...         css = {
...             "all": ["layout.css"],
...         }
...

>>> f = ContactForm()
>>> f.media
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<link href="https://static.example.com/layout.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>
<script src="https://static.example.com/whizbang.js"></script>
```