# django-angular Documentation

***Release 1.1.3***

**Jacob Rief**

**Sep 12, 2017**

# Contents

Django-Angular is a collection of utilities, which aim to ease the integration of Django with AngularJS by providing reusable components.

# CHAPTER 1

## Project's home

Check for the latest release of this project on Github.

Please report bugs or ask questions using the Issue Tracker.

CHAPTER 2

# Contents

## Installation

Install **django-angular**. The latest stable release can be found on PyPI

```
pip install django-angular
```

Change to the root directory of your project and install Node dependencies:

```
npm init
npm install angular@~1.5 --save
```

### Dependencies

**django-angular** has no dependencies to any other Django app, except `easy-thumbnails` and `Pillow` if using the image upload feature.

Projects using **django-angular** however require that AngularJS is installed through other means than pip. The best solution is to run:

```
npm install angularjs@1.5 --save
```

in your project's root folder and add it to the Django's static files search path:

```
npm install angularjs@1.5 --save

STATICFILES_DIRS = [
    ...
    ('node_modules', /path/to/my-project-root/node_modules'),
]
```

From the project's templates, you may refer the AngularJS files as:

```
<script src="{% static 'node_modules/angular/angular.js' %}" type="text/javascript">
```

### Configuration

Add `'djng'` to the list of `INSTALLED_APPS` in your project's `settings.py` file

```
INSTALLED_APPS = [
    ...
    'djng',
    'easy_thumbnails',  # optional, if ImageField is used
    ...
]
```

Don't forget to define your `STATIC_ROOT` and `STATIC_URL` properly. Since we load JavaScript and CSS files directly from our Node dependencies, add that directory to the static files search path:

```
STATICFILES_DIRS = [
    ('node_modules', os.path.join(PROJECT_DIR, 'node_modules')),
]
```

## Django-1.11

**Django**, since version 1.11, is shipped with an exchangeable widget rendering engine. This is a great improvement for **django-angular**, since it doensn't have to override the widgets and its renderers. Instead, your projects `settings.py`, please use this configuration directive:

```
FORM_RENDERER = 'djng.forms.renderers.DjangoAngularBootstrap3Templates'
```

if templates shall be rendered with a Bootstrap3 grid, otherwise use:

```
FORM_RENDERER = 'djng.forms.renderers.DjangoAngularTemplates'
```

---

**Note:** **django-angular** does not define any database models. It can therefore easily be installed without any database synchronization.

---

# Integrate AngularJS with Django

## XMLHttpRequest

As a convention in web applications, Ajax requests shall send the HTTP-Header:

```
X-Requested-With: XMLHttpRequest
```

while invoking POST-requests. In AngularJS versions 1.0.x this was the default behavior, but in versions 1.1.x this support has been dropped. Strictly speaking, Django applications do not require this header, but if it is missing, all invocations to:

```
request.is_ajax()
```

would return `False`, even for perfectly valid Ajax requests. Thus, if you use AngularJS version 1.1.x or later, add the following statement during module instantiation:

```
var my_app = angular.module('MyApp').config(function($httpProvider) {
    $httpProvider.defaults.headers.common['X-Requested-With'] = 'XMLHttpRequest';
});
```

## Template tags

Django and AngularJS share the same token for variable substitution in templates, ie. `{{ variable_name }}`. This should not be a big problem, since you are discouraged to mix Django template code with AngularJS template code. However, this recommendation is not viable in all situations. Sometime there might be the need to mix both template languages, one which is expanded by Django on the server, and one which is expanded by AngularJS in the browser.

The cleanest solution to circumvent this, is by using the verbatim tag, which became available in Django-1.5.

Another approach is to use the AngularJS mustaches inside a templatetag, for instance:

```
<h2>{% trans "The value you were looking for is: {{ my_data.my_value }}" %}</h2>
```

It is strongly discouraged to change the syntax of the AngularJS template tags, because it breaks the compatibility to all third party AngularJS directives, which are shipped with their own templates.

### Partials

In AngularJS, when used together with external templates, static HTML code often is loaded by a $templateCache. These so named partials can be placed in their own sub-directory below `STATIC_ROOT`.

If, for some reason you need mixed template code, ie. one which first is expanded by Django and later is parsed by AngularJS, then add a view such as

```
class PartialGroupView(TemplateView):
    def get_context_data(self, **kwargs):
        context = super(PartialGroupView, self).get_context_data(**kwargs)
        # update the context
        return context
```

Resolve this view in `urls.py`

```
partial_patterns = [
    url(r'^partial-template1.html$', PartialGroupView.as_view(template_name='partial-
↪template1.html'), name='partial_template1'),
    # ... more partials ...,
]

urlpatterns = [
    # ...
    url(r'^partials/', include(partial_patterns, namespace='partials')),
    # ...
]
```

By using the utility function

```
from djng.core.urlresolvers import urls_by_namespace

my_partials = urls_by_namespace('partials')
```

the caller obtains a list of all partials defined for the given namespace. This list can be used when creating a Javascript array of URL's to be injected into controllers or directives.

### Inlining Partials

An alternative method for handling AngularJS's partial code, is to use the special script type `text/ng-template` and mixing it into existing HTML code. Say, an AngularJS directive refers to a partial using `templateUrl: 'template/mixed-ng-snipped.html'` during the link phase, then that partial may be embedded inside a normal Django template using

```
<script id="template/mixed-ng-snipped.html" type="text/ng-template">
  <div>{{ resolved_by_django }}</div>
  <div>{% verbatim %}{{ resolved_by_angular }}{% endverbatim %}</div>
</script>
```

or if the `$interpolateProvider` is used to replace the AngularJS template tags

```
<script id="template/mixed-ng-snipped.html" type="text/ng-template">
  <div>{{ resolved_by_django }}</div>
  <div>{$ resolved_by_angular $}</div>
</script>
```

### Dynamically generated Javascript code

There might be good reasons to mix Django template with AngularJS template code. Consider a multilingual application, where text shall be translated, using the Django translation engine.

Also, sometimes your application must pass configuration settings, which are created by Django during runtime, such as reversing a URL. These are the use cases when to mix Django template with AngularJS template code. Remember, when adding dynamically generated Javascript code, to keep these sections small and mainly for the purpose of configuring your AngularJS module. **All other Javascript code must go into separate static files!**

> **Warning:** Never use Django template code to dynamically generate AngularJS controllers or directives. This will make it very hard to debug and impossible to add Jasmine unit tests to your code. Always do a clear separation between the configuration of your AngularJS module, which is part of *your* application, and the client side logic, which always shall be independently testable without the need of a running Django server.

## Bound Forms

AngularJS's does not consider bound forms, rather in their mindset data models shall be bound to the form's input fields by a controller function. This, for Django developers may be unfamiliar with their way of thinking. Hence, if bound forms shall be rendered by Django, the behavior of AngularJS on forms must be adopted using a special directive which overrides the built-in form directive.

To override the built-in behavior, refer to the Javascript file `django-angular.js` somewhere on your page

```
<script src="{% static 'djng/js/django-angular.min.js' %}" type="text/javascript"></
↪script>
```

and add the module dependency to your application initialization

```
var my_app = angular.module('myApp', [/* other dependencies */, 'djng.forms']);
```

# Running the demos

Shipped with this project, there are four demo pages, showing how to use the AngularJS validation and data-binding mechanisms in combination with Django forms. Use them as a starting point for your own application using **django-angular**.

To run the demos, change into the directory `examples` and start the development server:

```
pip install -r requirements.txt
./manage.py runserver
```

You can also run unit tests:

```
./manage.py test
```

Now, point a browser onto one of

- http://localhost:8000/classic_form/
- http://localhost:8000/form_validation/
- http://localhost:8000/model_scope/
- http://localhost:8000/combined_validation/
- http://localhost:8000/threeway_databinding/

## Classic Form

Classic Subscribe Form with no data validation.

## Form Validation

The *Form Validation* demo shows how to implement a Django form with enriched functionality to add AngularJS's form validation in a DRY manner. This demo combines the classes `NgFormValidationMixin` with Django's `forms.Form` . This demo works without an AngularJS controller.

## Model Form

The *Model Form* demo shows how to combine a Django form with `NgFormValidationMixin`, which creates an AngularJS model on the client in a DRY manner. This model, a Plain Old Javascript Object, then can be used inside an AngularJS controller for all kind of purposes. Using an XMLHttpRequest, this object can also be sent back to the server and bound to the same form is was created from.

## Model Form Validation

The *Model Form Validation* shows how to combined both techniques from above, to create an AngularJS model which additionally is validated on the client.

### Three-Way Data-Binding

*Three-Way Data-Binding* shows how to combine a Django form with `NgFormValidationMixin`, so that the form is synchronized by the server on all browsers accessing the same URL.

This demo is only available, if the external dependency Websocket for Redis has been installed.

### Artificial form constraints

These demos are all based on the same form containing seven different input fields: CharField, RegexField (twice), EmailField, DateField, IntegerField and FloadField. Each of those fields has a different constraint:

- *First name* requires at least 3 characters.
- *Last name* must start with a capital letter.
- *E-Mail* must be a valid address.
- *Phone number* can start with + and may contain only digits, spaces and dashes.
- *Birth date* must be a vaild date.
- *Weight* must be an integer between 42 and 95.
- *Height* must be a float value between 1.48 and 1.95.

Additionally there are two artificial constraints defined by the server side validation, which for obvious reasons require a HTTP round trip in order to fail. These are:

- The full name may not be "John Doe"
- The email address may not end in "@example.com", "@example.net" or similar.

If the client bypasses client-side validation by deactivating JavaScript, the server validation still rejects these error. Using form validation this way, incorrect form values always are rejected by the server.

## Integrate a Django form with an AngularJS model

When deriving from Django's `forms.Form` class in an AngularJS environment, it can be useful to enrich the rendered form output with an AngularJS HTML tag, such as:

```
ng-model="model_name"
```

where *model_name* corresponds to the named field from the declared form class.

### Sample code

Assume to have a simple Django form class with a single input field. Enrich its functionality by mixing in the **djng** class `NgModelFormMixin`

---

**Note:** Here the names **NgModelForm...** do not interrelate with Django's `forms.ModelForm`. Instead that name reflects the HTML attribute `ng-model` as used in `<form>`-elements under control of AngularJS.

---

```
from django import forms
from django.utils import six
from djng.forms import fields, NgDeclarativeFieldsMetaclass, NgModelFormMixin

class ContactForm(six.with_metaclass(NgDeclarativeFieldsMetaclass, NgModelFormMixin,␣
↪forms.Form)):
    subject = fields.CharField()
    # more fields ...
```

**Note:** Since **django-angular**-1.1, you must use the adopted field classes, instead of Django's own `fields`.

In the majority of cases, the Form is derived from Django's `forms.Form`, so the above example can be rewritten in a simpler way, by using the convenience class `NgForm` as a replacement:

```
from djng.forms import NgModelFormMixin, NgForm

class MyValidatedForm(NgModelFormMixin, NgForm):
    # members as above
```

If the Form shall inherit from Django's `forms.ModelForm`, use the convenience class `NgModelForm`:

```
from djng.forms import NgModelFormMixin, NgModelForm

class MyValidatedForm(NgModelFormMixin, NgModelForm):
    class Meta:
        model = Article

    # fields as usual
```

Now, each rendered form field gets an additional attribute `ng-model` containing the field's name. For example, the input field named `subject` now will be rendered as:

```
<input id="id_subject" type="text" name="subject" ng-model="subject" />
```

This means, that to a surrounding Angular controller, the field's value is immediately added to its `$scope`.

## Full working example

This demonstrates how to submit form data using an AngularJS controller. The Django view handling this unbound contact form class may look like

```
from django.views.generic import TemplateView

class ContactFormView(TemplateView):
    template = 'contact.html'

    def get_context_data(self, **kwargs):
        context = super(ContactFormView, self).get_context_data(**kwargs)
        context.update(contact_form=ContactForm())
        return context
```

with a template named `contact.html`:

```
<form ng-controller="MyFormCtrl" name="contact_form">
    {{contact_form}}
    <button ng-click="submit()">Submit</button>
</form>
```

and using some Javascript code to define the AngularJS controller:

```
my_app.controller('MyFormCtrl', function($scope, $http) {
    $scope.submit = function() {
        var in_data = { subject: $scope.subject };
        $http.post('/url/of/your/contact_form_view', in_data)
            .success(function(out_data) {
                // do something
            });
    }
});
```

Note that the `<form>` tag does not require any `method` or `action` attribute, since the promise `success` in the controller's submit function will handle any further action. The success handler, for instance could load a new page or complain about missing fields. It now it is even possible to build forms without using the `<form>` tag anymore. All what's needed from now on, is a working AngularJS controller.

As usual, the form view must handle the post data received through the POST (aka Ajax) request. However, AngularJS does not send post data using `multipart/form-data` or `application/x-www-form-urlencoded` encoding – rather, it uses plain JSON, which avoids an additional decoding step.

---

**Note:** In real code, do not hard code the URL into an AngularJS controller as shown in this example. Instead inject an object containing the URL into the form controller as explained in manage Django URL's for AngularJS

---

Add these methods to view class handling the contact form

```
import json
from django.views.decorators.csrf import csrf_exempt
from django.http import HttpResponseBadRequest

class ContactFormView(TemplateView):
    # use 'get_context_data()' from above

    @csrf_exempt
    def dispatch(self, *args, **kwargs):
        return super(ContactFormView, self).dispatch(*args, **kwargs)

    def post(self, request, *args, **kwargs):
        if not request.is_ajax():
            return HttpResponseBadRequest('Expected an XMLHttpRequest')
        in_data = json.loads(request.body)
        bound_contact_form = CheckoutForm(data={'subject': in_data.get('subject')})
        # now validate 'bound_contact_form' and use it as in normal Django
```

**Warning:** In real code, **do not** use the `@csrf_exempt` decorator, as shown here for simplicity. Please read on how to *protect your views from Cross Site Request Forgeries*.

### Prefixing the form fields

The problem with this implementation, is that one must remember to access each form field three times. Once in the declaration of the form, once in the Ajax handler of the AngularJS controller, and once in the post handler of the view. This make maintenance hard and is a violation of the DRY principle. Therefore it makes sense to add a prefix to the model names. One possibility would be to add the argument `scope_prefix` on each form's instantiation, ie.:

```
contact_form = ContactForm(scope_prefix='my_prefix')
```

This, however, has to be done across all instantiations of your form class. The better way is to hard code this prefix into the constructor of the form class

```
class ContactForm(NgModelFormMixin, forms.Form):
    # declare form fields

    def __init__(self, *args, **kwargs):
        kwargs.update(scope_prefix='my_prefix')
        super(ContactForm, self).__init__(*args, **kwargs)
```

Now, in the AngularJS controller, the scope for this form starts with an object named `my_prefix` containing an entry for each form field. This means that an input field, the is rendered as:

```
<input id="id_subject" type="text" name="subject" ng-model="my_prefix.subject" />
```

This also simplifies the Ajax submit function, because now all input fields are available as a single Javascript object, which can be posted as `$scope.my_prefix` to your Django view:

```
$http.post('/url/of/contact_form_view', $scope.my_prefix)
```

### Working with nested forms

**NgModelFormMixin** is able to handle nested forms as well. Just remember to add the attribute `prefix='subform_name'` with the name of the sub-form, during the instantiation of your main form. Now your associated AngularJS controller adds this additional model to the object `$scope.my_prefix`, keeping the whole form self-contained and accessible through one Javascript object, aka `$scope.my_prefix`.

The Django view responsible for handling the post request of this form, automatically handles the parsing of all bound form fields, even from the nested forms.

---

**Note:** Django, internally, handles the field names of nested forms by concatenating the prefix with the field name using a dash '-'. This behavior has been overridden in order to use a dot '.', since this is the natural separator between Javascript objects.

---

### Form with FileField or ImageField

If you have a `FileField` or an `ImageField` within your form, you need to provide a file upload handler. Please refer to the section *Upload Files and images Images* for details.

# Validate Django forms using AngularJS

Django's forms.Form class offers many possibilities to validate a given form. This, for obvious reasons is done on the server. However, customers may not always accept to submit a form, just to find out that they missed to input some correct data into a field. Therefore, adding client side form validation is a good idea and very common. But since client side validation easily can be bypassed, the same validation has to occur a second time, when the server accepts the forms data for final processing.

*This leads to code duplication and generally violates the DRY principle!*

## NgFormValidationMixin

A workaround to this problem is to use Django's form declaration to automatically generate client side validation code, suitable for AngularJS. By adding a special mixin class to the form class, this can be achieved automatically and on the fly

```
from django import forms
from django.utils import six
from djng.forms import fields, NgDeclarativeFieldsMetaclass, NgFormValidationMixin

class MyValidatedForm(six.with_metaclass(NgDeclarativeFieldsMetaclass,␣
↪NgFormValidationMixin, forms.Form)):
    form_name = 'my_valid_form'
    surname = fields.CharField(label='Surname', min_length=3, max_length=20)
    age = fields.DecimalField(min_value=18, max_value=99)
```

**Note:** Since **django-angular**-1.1, you must use the adopted field classes, instead of Django's own `fields`.

In the majority of cases, the Form is derived from Django's `forms.Form`, so the above example can be rewritten in a simpler way, by using the convenience class `NgForm` as a replacement:

```
from djng.forms import NgFormValidationMixin, NgForm

class MyValidatedForm(NgFormValidationMixin, NgForm):
    # members as above
```

If the Form shall inherit from Django's `forms.ModelForm`, use the convenience class `NgModelForm`:

```
from djng.forms import NgFormValidationMixin, NgModelForm

class MyValidatedForm(NgFormValidationMixin, NgModelForm):
    class Meta:
        model = Article

    # fields as usual
```

Each page under control of AngularJS requires a unique form name, otherwise the AngularJS's form validation engine shows undefined behavior. Therefore you must name each form inheriting from `NgFormValidationMixin`. If a form is used only once per page, the form's name can be added to the class declaration, as shown above. If no form name is specified, it defaults to `form`, limiting the number of validated forms per page to one.

If a form inheriting from `NgFormValidationMixin` shall be instantiated more than once per page, each instance of that form must be instantiated with a different name. This then must be done in the constructor of the form, by passing in the argument `form_name='my_form'`.

In the view class, add the created form to the rendering context:

```
def get_context_data(self, **kwargs):
    context = super(MyRenderingView, self).get_context_data(**kwargs)
    context.update(form=MyValidatedForm())
    return context
```

or if the same form declaration shall be used more than once:

```
def get_context_data(self, **kwargs):
    context = super(MyRenderingView, self).get_context_data(**kwargs)
    context.update(form1=MyValidatedForm(form_name='my_valid_form1'),
                   form2=MyValidatedForm(form_name='my_valid_form2'))
    return context
```

**Note:** Do not use an empty `label` when declaring a form field, otherwise the class `NgFormValidationMixin` won't be able to render AngularJS's validation error elements. This also applies to `auto_id`, which if False, will not include `<label>` tags while rendering the form.

### Render this form in a template

```
<form name="{{ form.form_name }}" novalidate>
  {{ form }}
  <input type="submit" value="Submit" />
</form>
```

Remember to add the entry `name="{{ form.form_name }}"` to the `form` element, otherwise AngularJS's validation engine won't work. Use the directive `novalidate` to disable the browser's native form validation. If you just need AngularJS's built in form validation mechanisms without customized checks on the forms data, there is no need to add an `ng-controller` onto a wrapping HTML element. The only measure to take, is to give each form on a unique name, otherwise the AngularJS form validation engine shows undefined behavior.

Forms which do not validate on the client, probably shall not be posted. This can simply be disabled by replacing the submit button with the following HTML code:

```
<input type="submit" class="btn" ng-disabled="{{ form.form_name }}.$invalid" value=
↪"Submit">
```

### More granular output

If the form fields shall be explicitly rendered, the potential field validation errors can be rendered in templates using a special field tag. Say, the form contains

```
from django import forms
from djng.forms import fields, NgFormValidationMixin

class MyValidatedForm(NgFormValidationMixin, forms.Form):
        email = fields.EmailField(label='Email')
```

then access the potential validation errors in templates using `{{ form.email.errors }}`. This renders the form with an unsorted list of potential errors, which may occur during client side validation.

```
<ul class="djng-form-errors" ng-hide="subscribe_form.email.$pristine">
  <li ng-show="subscribe_form.email.$error.required" class="ng-hide">This field is␣
↪required.</li>
  <li ng-show="subscribe_form.email.$error.email" class="">Enter a valid email␣
↪address.</li>
</ul>
```

The AngularJS form validation engine, normally hides these potential errors. They only become visible, if the user enters an invalid email address.

### Bound forms

If the form is bound and rendered, then errors detected by the server side's validation code are rendered as unsorted list in addition to the list of potential errors. Both of these error lists are rendered using their own `<ul>` elements. The behavior for potential errors remains the same, but detected errors are hidden the moment, the user sets the form into a dirty state.

---

**Note:** AngularJS normally hides the content of bound forms, which means that `<input>` fields seem empty, even if their `value` attribute is set. In order to restore the content of those input fields to their default value, initialize your AngularJS application with `angular.module('MyApp', ['djng.forms']);`.

---

### Combine NgFormValidationMixin with NgModelFormMixin

While it is possible to use `NgFormValidationMixin` on itself, it is perfectly legal to mix `NgModelFormMixin` with `NgFormValidationMixin`. However, a few precautions have to be taken.

On class declaration inherit first from `NgModelFormMixin` and *afterward* from `NgFormValidationMixin`. Valid example:

```
from django import forms
from djng.forms import NgFormValidationMixin, NgModelFormMixin

class MyValidatedForm(NgModelFormMixin, NgFormValidationMixin, forms.Form):
    # custom form fields
```

but don't do this

```
class MyValidatedForm(NgFormValidationMixin, NgModelFormMixin, forms.Form):
    # custom form fields
```

Another precaution to take, is to use different names for the forms name and the `scope_prefix`. So, this is legal

```
form = MyValidatedForm(form_name='my_form', scope_prefix='my_model')
```

but this is not

```
form = MyValidatedForm(form_name='my_form', scope_prefix='my_form')
```

### An implementation note

AngularJS names each input field to validate, by concatenating its forms name with its fields name. This object member then contains an error object, named `my_form.field_name.$error` filled by the AngularJS validation

mechanism. The placeholder for the error object would clash with `ng-model`, if the form name is identical to the scope prefix. Therefore, just remember to use different names.

## Customize detected and potential validation errors

If a form with AngularJS validation is rendered, each input field is prefixed with an unsorted list `<ul>` of potential validation errors. For each possible constraint violation, a list item `<li>` containing a descriptive message is added to that list.

If a client enters invalid data into that form, AngularJS unhides one of these prepared error messages, using ng-show. The displayed message text is exactly the same as would be shown if the server side code complains about invalid data during form validation. These prepared error messages can be customized during form field definition.

The default error list is rendered as `<ul class="djng-form-errors">...</ul>`. To each `<li>` of this error list, the attribute `class="invalid"` is added. The last list-item `<li class="valid"></li>` is somehow special, as it is only visible if the corresponding input field contains valid data. By using special style sheets, one can for instance add a green tick after a validated input field, to signal that everything is OK.

The styling of these validation elements must be done through CSS, for example with:

```
ul.djng-form-errors {
        margin-left: 0;
        display: inline-block;
        list-style-type: none;
}
ul.djng-form-errors li.invalid {
        color: #e9322d;
}
ul.djng-form-errors li.invalid:before {
        content: "\2716\20";  /* adds a red cross before the error message */
}
ul.djng-form-errors li.valid:before {
        color: #00c900;
        content: "\2714";  /* adds a green tick */
}
```

If you desire an alternative CSS class or an alternative way of rendering the list of errors, then initialize the form instance with

```
class MyErrorList(list):
    # rendering methods go here

# during form instantiation
my_form = MyForm(error_class=MyErrorList)
```

Refer to `TupleErrorList` on how to implement an alternative error list renderer. Currently this error list renderer, renders two `<ul>`-elements for each input field, one to be shown for *pristine* forms and one to be shown for *dirty* forms.

### Adding an AngularJS directive for validating form fields

Sometimes it can be useful to add a generic field validator on the client side, which can be controlled by the form's definition on the server. One such example is Django's DateField:

```
from django import forms
```

```
class MyForm(forms.Form):
    # other fields
    date = forms.DateField(label='Date',
        widget=forms.widgets.DateInput(attrs={'validate-date': '^(\d{4})-(\d{1,2})-(\d
↪{1,2})$'}))
```

Since AngularJS can not validate dates, such a field requires a customized directive, which with the above definition, will be added as new attribute to the input element for date:

```
<input name="date" ng-model="my_form_data.birth_date" type="text" validate-date="^(\d
↪{4})-(\d{1,2})-(\d{1,2})$" />
```

If your AngularJS application has been initialized with

```
angular.module('MyApp', ['djng.forms']);
```

then this new attribute is detected by the AngularJS directive `validateDate`, which in turn checks the date for valid input and shows the content of the errors fields, if not.

If you need to write a reusable component for customized form fields, refer to that directive as a starting point.

# Tutorial: Django Forms with AngularJS

Django offers an excellent Form framework which is responsible for rendering and validating HTML forms. Since Django's design philosophy is to be independent of the styling and JavaScript, this Form framework requires some adoption in order to play well with AngularJS and optionally Bootstrap.

A common technique to adopt the styling of a Django Form, is to extend the template so that each Form field is rendered by hand written HTML. This of course leads to code duplication and is a violation of the DRY principle.

An alternative technique is to add crispy-forms to your project and enrich the Django Forms with a helper class. Unfortunately, **crispy-form** does not work very well with **django-angular**. In order to add the same functionality in a "DRY" manner, a special Mixin class can be added to your forms, rather than having to work with helper-classes.

This tutorial attempts to explain how to combine the Django Form framework in combination with AngularJS and Bootstrap.

## Basic Form Submission

Lets start with a very basic example, a functioning demo is available here: http://django-angular.awesto.com/classic_form/

Say, we have a simple but rather long Form definition, to subscribe a person wanting to visit us:

Since we want to render this form in a DRY manner, our favorite template syntax is something such as:

```
<form name="{{ form.form_name }}" method="post" action=".">
        {% csrf_token %}
        {{ form }}
        <input type="submit" value="Subscribe">
</form>
```

In this example, the whole form is rendered in one pass including all HTML elements, such as `<label>`, `<select>`, `<option>` and `<input>`. Additionally, bound forms are rendered with their preset values and a list of errors, if the previous Form validation did not succeed.

The Django Form framework comes with three different rendering methods: `as_p()`, `as_ul()` and `as_table()` (the default). Unfortunately, these three rendering methods are not suitable for nowadays needs, such as Bootstrap styling and Ajax based Form validation.

In order to be more flexible without having to abandon the "DRY" principle, the above `SubscriptionForm` has been enriched by the Mixin class `Bootstrap3FormMixin`. This class adds an additional method `as_div()`, which is responsible for rendering the Form suitable for Bootstrap styling.

Additionally, this Mixin class wraps the list of validation errors occurred during the last Form validation into the AngularJS directive `ng-show`, so that error messages disappear after the user starts typing, and thus puts the Form into a "dirty", or in other words "non-pristine", state.

You can test this yourself, by leaving out some fields or entering invalid values and submitting the Form.

### Bound Form in AngularJS

AngularJS does not take into account the concept of bound Forms. Therefore, input fields rendered with preset values, are displayed as empty fields. To circumvent this, the **django-angular** Form directive re-enables the rendering of the bound field values.

### Dynamically Hiding Form Fields for Bootstrap

A common use case is to hide a form field based on the value of another. For example, to hide the `phone` field if the user selects *Female* within `SubscriptionForm`, overwrite `field_css_classes` on `SubscriptionForm`:

```
field_css_classes = {
    '*': 'form-group has-feedback',
    'phone': "ng-class:{'ng-hide':sex==='f'}",
}
```

`field_css_classes` adds css classes to the wrapper div surrounding individual fields in bootstrap. In the above example, `'*'` adds the classes `form-group has-feedback` to all fields within the form and `'ng-class:{"ng-hide":sex==="f"}'` is added only to the `phone` field. Only Angular directives that can be used as CSS classes are allowed within `field_css_classes`. Additionally, if specified as a string, the string may not contain any spaces or double quotes. However, if specified as a list, spaces can be used, and the above example can be rewritten as:

```
field_css_classes = {
    '*': 'form-group has-feedback',
    'phone': ["ng-class: {'ng-hide': sex==='f'};"],
}
```

By adding the keyword `'__default__'` to this list, the CSS classes for the default entry, ie. `'*'`, are merged with the CSS classes for the current field.

### Adding an asterisk for required fields

An asterisk is often added after labels on required fields (like with django-crispy-forms for example). This can be accomplished by using native Django and CSS. To do this, set the `required_css_class attribute` on `SubscriptionForm` (see Django documentation).

```
required_css_class = 'djng-field-required'
```

Next, add the *CSS*:

```
label.djng-field-required::after {
    content: "\00a0*";
}
```

### Client-side Form validation

To enable client-side Form validation, simply add the mixin class `NgFormValidationMixin` to the `SubscriptionForm` class:

```
class SubscriptionForm(NgFormValidationMixin, Bootstrap3FormMixin, forms.Form):
    # form fields as usual
```

Here the rendered Form contains all the AngularJS directives as required for client side Form validation. If an entered value does not match the criteria as defined by the definition of `SubscriptionForm`, AngularJS will notify the user immediately

## Upload Files and images Images

**Django-Angular** emphasizes the use of Ajax request-response cycles while handling form data. One disadvantage of this approach is, that you can't use it to upload files to the server, because browsers can not serialize file payload into JSON. Instead, in order to upload files one must **POST** a `<form>` using `enctype="multipart/form-data"`.

This approach nowadays is outdated. Moreover, it requires the use of an `<input type="file">` field, which doesn't provide a good user experience either. A disfunctional example:

Instead, we nowadays are used to drag files directly into the browser window and drop them onto an input field, which immediately displays the uploaded image. By adding two third party packages, **django-angular** provides such a solution.

By replacing Django's form fields `FileField` against `djng.forms.fields.FileField` and `ImageField` against `djng.forms.fields.ImageField`, the corresponding form field is rendered as a rectangular area, where one can drag a file or image onto, and drop it. It then is uploaded immediately to the server, which keeps it in a temporary folder and returns a thumbail of that file/image together with a reference onto a temporary representation.

In the next step, when the user submits the form, only the reference to that temporary file is added to the post data. Therefore the payload of such a form can be posted using JSON via Ajax. This gives a much smoother user experience, rather than uploading the form together with the image payload using a full request-response cycle.

### Installation

```
pip install easy-thumbnails
```

and add it to the project's `settings.py`:

```
INSTALLED_APPS = [
    ...
    'djng',
    'easy_thumnails',
    ...
]
```

Check that your MEDIA_ROOT points onto a writable directory. Use MEDIA_URL = '/media/' or whatever is appropriate for your project.

Install additional Node dependencies:

```
npm install ng-file-upload --save
```

## Usage in Forms

First, we must add an endpoint to our application which receives the uploaded images. To the urls.py of the project add:

```
from djng.views.upload import FileUploadView

urlpatterns = [
    ...
    url(r'^upload/$', FileUploadView.as_view(), name='fileupload'),
    ...
]
```

By default files are uploaded into the directory <MEDIA_ROOT>/upload_temp. This location can be changed using the settings variable DJNG_UPLOAD_TEMP.

In our form declaration, we replace Django's ImageField by an alternative implementation provided by **django-angular**. This class accepts two optional additional attributes:

- fileupload_url: The URL pointing onto the view accepting the uploaded image. If omitted, it defaults to the URL named fileupload.
- area_label: This is the text rendered inside the draggable area. Don't confuse this with the label, which is rendered before that area.

An example:

```
from django.core.urlresolvers import reverse_lazy
from djng.forms import NgModelFormMixin
from djng.forms.fields import ImageField
from . import subscribe_form

class SubscribeForm(NgModelFormMixin, subscribe_form.SubscribeForm):
    scope_prefix = 'my_data'
    form_name = 'my_form'

    photo = ImageField(
        label='Photo of yourself',
        fileupload_url=reverse_lazy('fileupload'),
        area_label='Drop image here or click to upload',
        required=True)
```

The Django View responsible for accepting submissions from that form, works just as if Django's internal django.forms.fields.ImageField would have been used. The attribute cleaned_data['photo'] then contains an object of type FieldFile after a form submission.

## Usage in Models

Often you might use a model and rely on Django's automatic form generation. **django-angular** does this out-of-the-box, whenever the form implementing the model inherits form NgModelForm.

### Usage in Templates

When using this file uploader, the Angular App requires an additional stylesheet and an external JavaScript module:

```
{% load static %}

<head>
    ...
    <link href="{% static 'djng/css/fileupload.css' %}" rel="stylesheet" />
</head>

<body>
    ...
    <script src="{% static 'node_modules/ng-file-upload/dist/ng-file-upload.js' %}"␣
↪type="text/javascript"></script>
    <script src="{% static 'djng/js/django-angular.min.js' %}" type="text/javascript">
↪</script>
</body>
```

additionally, the Angular App must be initialized such as:

```
<script>
angular.module('myApp', [..., 'djng.fileupload', 'djng.forms', ...])
.config(['$httpProvider', function($httpProvider) {
    $httpProvider.defaults.headers.common['X-Requested-With'] = 'XMLHttpRequest';
    $httpProvider.defaults.headers.common['X-CSRFToken'] = '{{ csrf_token }}';
}]);
</script>
```

### Caveats

When users upload images, but never submit the corresponding form, the folder holding these temporary images gets filled up. Therefore you should add some kind of (cron)job which cleans up that folder from time to time.

Depending on your setup, also provide some security measure, so that for example, only logged in users have access onto the view for uploading images. Otherwise the temporary folder might get filled with crap.

### Security Measures

Altought the relative location of the uploaded files is returned to the client and visible in its browser, it almost is impossible to access images which have not been uploaded by the provided class `djng.views.FileUploadView`, or rendered by the provided widget `djng.forms.widgets.DropFileInput`. This is because all file names are cryptographically signed, so to harden them against tampering. Otherwise someone else could pilfer or delete images uploaded to the temporary folder, provided that he's able to guess the image name.

## Perform basic CRUD operations

When using Angular's $resource to build services, each service comes with free CRUD (create, read, update, delete) methods:

```
{ 'get':    {method:'GET'},
  'save':   {method:'POST'},
  'query':  {method:'GET', isArray:true},
```

```
  'remove': {method:'DELETE'},
  'delete': {method:'DELETE'}
};
```

Of course this need support on the server side. This can easily be done with **django-angular**'s `NgCRUDView`.

---

**Note:** `remove()` and `delete()` do exactly the same thing. Usage of `remove()` is encouraged, since `delete` is a reserved word in IE.

---

## Configuration

Subclass `NgCRUDView` and override model attribute:

```
from djng.views.crud import NgCRUDView

class MyCRUDView(NgCRUDView):
    model = MyModel
```

Add urlconf entry pointing to the view:

```
...
url(r'^crud/mymodel/?$', MyCRUDView.as_view(), name='my_crud_view'),
...
```

Set up Angular service using `$resource`:

```
var myServices = angular.module('myServices', ['ngResource']);

myServices.factory('MyModel', ['$resource', function($resource) {
    return $resource('/crud/mymodel/', {'pk': '@pk'}, {
    });
}]);
```

---

**Note:** Since there is a known bug with $resource not respecting trailing slash, the urls in Django urlconf used by $resource must either not have trailing slash or it should be optional (preferred) - e.g. `url/?`. Adding the trailing slash to the $resource configuration regardless (`/crud/mymodel/`) ensures future compatibility in case the bug gets fixed and will then follow Django's trailing slash convention. This has been fixed in AngularJS 1.3. More information here trailingSlashBugFix

---

Another quick change is required to Angular app config, without this `DELETE` requests fail `CSRF` test:

```
var my_app = angular.module('myApp', [/* other dependencies */, 'ngCookies']).run(
    function($http, $cookies) {
        $http.defaults.headers.post['X-CSRFToken'] = $cookies.csrftoken;
        // Add the following two lines
        $http.defaults.xsrfCookieName = 'csrftoken';
        $http.defaults.xsrfHeaderName = 'X-CSRFToken';
    });
```

That's it. Now you can use CRUD methods.

## Optional attributes

The following options are currently available to subclasses of `NgCRUDView`:

#### fields

Set this to a tuple or list of field names for only retrieving a subset of model fields during a *get* or *query* operation. Alternatively, if this may vary (e.g. based on query parameters or between *get* and *query*) override the `get_fields()` method instead.

With `None` (default), all model fields are returned. The object identifier (`pk`) is always provided, regardless of the selection.

#### form_class

Set this to a specific form for your model to perform custom validation with it. Alternatively, if it may vary you can override the `get_form_class()` method instead.

With `None` (default), a modelForm including all fields will be generated and used.

#### slug_field

Similar to Django's SingleObjectMixin, objects can be selected using an alternative key such as a title or a user name. Especially when using the ngRoute module of AngularJS, this makes construction of descriptive URLs easier. Query parameters can be extracted directly from $route or $routeParams and passed to the query.

This attribute (default is `'slug'`) describes the field name in the model as well as the query parameter from the client. For example, if it is set to `'name'`, perform a query using

```
var model = MyModel.get({name: "My name"});
```

---

**Note:** Although the view will not enforce it, it is strongly recommended that you only use unique fields for this purpose. Otherwise this can lead to a `MultipleObjectsReturned` exception, which is not handled by this implementation.

Also note that you still need to pass the object identifier `pk` on update and delete operations. Whereas for save operations, the check on `pk` makes the distinction between an update and a create operation, this restriction on deletes is only for safety purposes.

---

#### allowed_methods

By default, `NgCRUDView` maps the request to the corresponding django-angular method, e.g. a `DELETE` request would call the `ng_delete` method.

`allowed_methods` is set by default to `['GET', 'POST', 'DELETE']`. If you need to prevent any method, you can overrride the `allowed_methods` attributes. Alternatively, you can use the `exclude_methods` attributes.

#### exclude_methods

To allow all methods by default, `exclude_methods` is set as an empty list. To exclude any method, you can override this attribute to exclude the `'GET'`, `'POST'` or `'DELETE'`. See `allowed_methods` for more informations.

### Usage example

```
myControllers.controller('myCtrl', ['$scope', 'MyModel', function ($scope, MyModel) {
    // Query returns an array of objects, MyModel.objects.all() by default
    $scope.models = MyModel.query();

    // Getting a single object
    var model = MyModel.get({pk: 1});


    // We can crete new objects
    var new_model = new MyModel({name: 'New name'});
    new_model.$save(function(){
       $scope.models.push(new_model);
    });
    // In callback we push our new object to the models array

    // Updating objects
    new_model.name = 'Test name';
    new_model.$save();

    // Deleting objects
    new_model.$remove();
    // This deletes the object on server, but it still exists in the models array
    // To delete it in frontend we have to remove it from the models array

}]);
```

**Note:** In real world applications you might want to restrict access to certain methods. This can be done using decorators, such as `@login_required`. For additional functionality *JSONResponseMixin* and `NgCRUDView` can be used together.

## Remote Method Invocation

Wouldn't it be nice to call a Django view method, directly from an AngularJS controller, similar to a Remote Procedure Call or say better **Remote Method Invocation**?

### Single Page Applications

By nature, Single Page Web Applications implemented in Django, require one single View. These kind of applications can however not always be build around the four possible request methods `GET`, `PUT`, `POST` and `DELETE`. They rather require many different entry points to fulfill the communication between the client and the server.

Normally, this is done by adding a key to the request data, which upon evaluation calls the appropriate method. However, such an approach is cumbersome and error-prone.

*Django-Angular* offers some helper functions, which allows the client to call a Django's View method, just as if it would be a normal asynchronous JavaScript function. To achieve this, let the View's class additionally inherit from JSONResponseMixin:

```
from django.views.generic import View
from djng.views.mixins import JSONResponseMixin, allow_remote_invocation
```

```
class MyJSONView(JSONResponseMixin, View):
    # other view methods

    @allow_remote_invocation
    def process_something(self, in_data):
        # process in_data
        out_data = {
            'foo': 'bar',
            'success': True,
      }
      return out_data
```

In this Django View, the method `process_something` is decorated with `@allow_remote_invocation`. It now can be invoked directly from an AngularJS controller or directive. To handle this in an ubiquitous manner, *Django-Angular* implements two special template tags, which exports *all* methods allowed for remote invocation to the provided AngularJS service `djangoRMI`.

### Template Tag `djng_all_rmi`

The AngularJS Provider `djangoRMIProvider` shall be configured during the initialization of the client side, such as:

```
{% load djng_tags %}
...
<script type="text/javascript">
var tags = {% djng_all_rmi %};
my_app.config(function(djangoRMIProvider) {
    djangoRMIProvider.configure(tags);
});
</script>
```

This makes available *all* methods allowed for remote invocation, from *all* View classes of your Django project.

**Note:** In order to have your methods working, the associated urls need to be named.

### Template Tag `djng_current_rmi`

Alternatively, the AngularJS Provider `djangoRMIProvider` can be configured during the initialization of the client side, such as:

```
{% load djng_tags %}
...
<script type="text/javascript">
var tags = {% djng_current_rmi %};
my_app.config(function(djangoRMIProvider) {
    djangoRMIProvider.configure(tags);
});
</script>
```

This makes available *all* methods allowed for remote invocation, from the current View class, ie. the one rendering the current page.

**Note:** In order to have your methods working, the associated urls need to be named.

### Let the client invoke an allowed method from a Django View

By injecting the service `djangoRMI` into an AngularJS controller, allowed methods from the Django View which renders the current page, can be invoked directly from JavaScript. This example shows how to call the above Python method `process_something`, when configured using the template tag `djng_current_rmi`:

```
my_app.controller("SinglePageCtlr", function($scope, djangoRMI) {
    $scope.invoke = function() {
        var in_data = { some: 'data' };
        djangoRMI.process_something(in_data)
            .success(function(out_data) {
                // do something with out_data
            });
    };
});
```

If `djangoRMIProvider` is configured using the template tag `djng_all_rmi`, the allowed methods are grouped into objects named by their url_name. If these URL patterns are part of a namespace, the above objects furthermore are grouped into objects named by their namespace.

**Note:** djangoRMI is a simple wrapper around AngularJS's built in $http service. However, it automatically determines the correct URL and embeds the method name into the special HTTP-header `DjNg-Remote-Method`. In all other aspects, it behaves like the $http service.

## Dispatching Ajax requests using method GET

Sometimes you only have to retrieve some data from the server. If you prefer to fetch this data using an ordinary GET request, ie. one without the special AngularJS provider `djangoRMI`, then it is possible to hard-code the method for invocation into the urlpatterns inside the URL dispatcher.

```
class MyResponseView(JSONResponseMixin, View):
    def get_some_data(self):
        return {'foo': 'bar'}

    def get_other_data(self):
        return ['baz', 'cap']

urlpatterns = [
    ...
    url(r'^fetch-some-data.json$', MyResponseView.as_view(), {'invoke_method': 'get_
↪some_data'}),
    url(r'^fetch-other-data.json$', MyResponseView.as_view(), {'invoke_method': 'get_
↪other_data'}),
    ...
]
```

If a client calls the URL `/fetch-some-data.json`, the responding view dispatches incoming requests directly onto the method `get_some_data`. This kind of invocation only works for GET requests. Here these methods *do*

*not* require the decorator `@allow_remote_invocation`, since now the server-side programmer is responsible for choosing the correct method and thus a malicious client cannot bypass the intended behavior.

# Cross Site Request Forgery protection

Ajax requests submitted using method POST are put to a similar risk for Cross Site Request Forgeries as HTTP forms. This type of attack occurs when a malicious Web site is able to invoke an Ajax request onto your Web site. In Django, one should always add the template tag csrf_token to render a hidden input field containing the token, inside each form submitted by method POST.

When it comes to making an Ajax request, it normally is not possible to pass that token using a Javascript object, because scripts usually are static and no secret can be added dynamically. AngularJS natively supports CSRF protection, only some minor configuration is required to work with Django.

## Configure Angular for Django's CSRF protection

Angular looks for `XSRF-TOKEN` cookie and submits it in `X-XSRF-TOKEN` http header, while Django sets `csrftoken` cookie and expects `X-CSRFToken` http header. All we have to do is change the name of cookie and header Angular uses. This is best done in `config` block:

```
var my_app = angular.module('myApp', [/* dependencies */]).config(function(
↪$httpProvider) {
    $httpProvider.defaults.xsrfCookieName = 'csrftoken';
    $httpProvider.defaults.xsrfHeaderName = 'X-CSRFToken';
});
```

When using this approach, ensure that the `CSRF` cookie is *not* configured as HTTP_ONLY, otherwise for security reasons that value can't be accessed from JavaScript.

Alternatively, if the block used to configure the AngularJS application is rendered using a Django template, one can add the value of the token directly to the request headers:

```
var my_app = angular.module('myApp', [/* dependencies */]).config(function(
↪$httpProvider) {
    $httpProvider.defaults.headers.common['X-CSRFToken'] = '{{ csrf_token }}';
    $httpProvider.defaults.headers.common['X-Requested-With'] = 'XMLHttpRequest';
}
```

# Share a template between Django and AngularJS

Templates syntax for Django and AngularJS is very similar, and with some caveats it is possible to reuse a Django template for rendering in AngularJS. The classical approach to embed AngularJS template code inside Django's template code, is to use the `{% verbatim %}` template tag. This tag however deactivates all Django's template parsing, so every block tag must be placed outside a `{% verbatim %}` ... `{% endverbatim %}` section. This makes mixed template coding quite messy.

## For this purpose use the template tag `{% angularjs %}`

The template tag `{% angularjs %}` ... `{% endangularjs %}` delegates Django's variable expansion to AngularJS, but continues to process the Django block tags, such as `{% if ... %}`, `{% for ... %}`, `{% load ... %}`, etc.

### Conditionally activate variable expansion

The template tag `{% angularjs <arg> %}` takes one optional argument, which when it evaluates to true, it turns on AngularJS's variable expansion. Otherwise, if it evaluates to false, it turns on Django's variable expansion. This becomes handy when using include snippets which then can be used by both, the client and the server side template rendering engines.

## Example

A Django ListView produces a list of items and this list is serializable as JSON. For browsers without JavaScript and for crawlers from search engines, these items shall be rendered through the Django's template engine. Otherwise, AngularJS shall iterate over this list and render these items.

Template used by the list view:

```
<div ng-if="!items">
{% for item in items %}
    {% include "path/to/includes/item-detail.html" with ng=0 %}
{% endfor %}
</div>
<div ng-if="items">
{% include "path/to/includes/item-detail.html" with ng=1 %}
</div>
```

Here the scope variable `items` is set by a surrounding `ng-controller`. As we can see, the template `path/to/includes/item-detail.html` is included twice, once defining an additional context variable `ng` as true and later, the same include with that variable as false.

Assume, this list view shall render a model, which contains the following fields:

```
class Item(models.Model):
    title = CharField(max_length=50)
    image = ImageField()  # built-in or from a third party library
    description = HTMLField()  # from a third party library

    def get_absolute_url(self):
        return reverse(...)
```

Now, the included template can be written as:

```
{% load djng_tags %}
{% angularjs ng %}
<div{% if ng %} ng-repeat="item in items"{% endif %}>
    <h4><a ng-href="{{ item.absolute_url }}"{% if not ng %} href="{{ item.absolute_
↪url }}"{% endif %}>{{ item.name }}</a></h4>
    <img ng-src="{{ item.image.url }}"{% if not ng %} src="{{ item.image.url }}"{%␣
↪endif %} width="{{ item.image.width }}" height="{{ item.image.height }}" />
    <div{% if ng %} ng-bind-html="item.description"{% endif %}>{% if not ng %}{{ item.
↪description }}{% endif %}</div>
</div>
{% endangularjs %}
```

A few things to note here:

The content between the template tags `{% angularjs ng %}` and `{% endangularjs %}` is rendered through the Django template engine as usual, if the context variable `ng` evaluates to false. Otherwise all variable expansions,

ie. `{{ varname }}` or `{{ varname|filter }}` are kept as-is in HTML, while block tags are expanded by the Django template engine.

The context data, as created by the list view, must be processed into a list serializable as JSON. This list then can be used directly by the Django template renderer or transferred to the AngularJS engine, using a XMLHttpRequest or other means.

This means that the default method `get_context_data()` must resolve all object fields into basic values, since invocations to models methods, such as `get_absolute_url()`, now can not be done by the template engine, during the iteration step, ie. `{% for item in items %}`. The same applies for image thumbnailing, etc.

In AngularJS references onto URLs and image sources must be done with `<a ng-href="...">` and `<img ng-src="...">`, rather than using `<a href="...">` or `<img src="...">` respectively. Therefore, while rendering the Django template, these fields are added twice.

In AngularJS, text data containing HTML tags, must be rendered using ng-bind-html rather than using the mustache syntax. This is to ensure, that unverified content from upstream sources is sanitized. We can assert this, since this text content is coming from the database field `description` and thus is marked as safe string by Django.

### Python List / Javascript Arrays

The Django template engine accesses members of Python dictionaries using the *dot* notation. This is the same notation as used by JavaScript to access members of objects. When accessing lists in Django templates or arrays in JavaScript, this notation is not compatible any more. Therefore as convenience, always use the Django template notation, even for JavaScript arrays. Say, in Python you have a list of objects:

```
somelist = [{'member': 'first'}, {'member': 'second'}, {'member': 'third'},]
```

To access the third member, Django's template code shall be written as:

```
{{ somelist.2.member }}
```

when this block is resolved for AngularJS template rendering, the above code is expanded to:

```
{{ somelist[2].member }}
```

otherwise it would be impossible to reuse Python lists converted to JavaScript arrays inside the same template code.

### Conditionally bind scope variables to an element with `djng-bind-if`

Sometimes it makes sense to bind the scope variable to an element if it exists. Otherwise render the same variable from Django's context. Example:

```
<span djng-bind-if="some_prefix.value">{{ some_prefix.value }}</span>
```

functionally, this is equivalent to:

> {% verbatim %}{{ some_prefix.value }}{% endverbatim %}
> {{ some_prefix.value }}

but less verbose and easier to read.

# Manage Django URLs for AngularJS

AngularJS controllers communicating with the Django application through Ajax, often require URLs, pointing to some of the views of your application. Don't fall into temptation to hard code such a URL into the client side controller code. Even worse would be to create Javascript dynamically using a template engine. There is a clean and simple solution to solve this problem.

---

**Note:** Until version 0.7.14 **django-angular** reversed all existing URLs of a project and created an object exposing them to Javascript. Documentation for now deprecated approach is available here.

Starting with version 0.7.15, **django-angular** provides a new way to handle URLs, which offers the reversing functionality directly to AngularJS modules.

---

This service is provided by `djangoUrl.reverse(name, args_or_kwargs)` method. It behaves exactly like Django's URL template tag.

## Basic operation principle

**django-angular** encodes the parameters passed to `djangoUrl.reverse()` into a special URL starting with `/angular/reverse/...`. This URL is used as a new entry point for the real HTTP invocation.

## Installation

### Angular

- Include `django-angular.js`:

```
<script src="{% static 'djng/js/django-angular.js' %}"></script>
```

- Add `djng.urls` as a dependency for you app:

```
<script>
    var my_app = angular.module('MyApp', ['djng.urls', /* other dependencies */]);
</script>
```

The `djangoUrl` service is now available through dependency injection to all directives and controllers.

### Setting via Django Middleware

- Add `'djng.middleware.AngularUrlMiddleware'` to `MIDDLEWARE_CLASSES` in your Django `settings.py` file:

```
MIDDLEWARE_CLASSES = (
    'djng.middleware.AngularUrlMiddleware',
    # Other middlewares
)
```

**Warning:** This must be the **first** middleware included in `MIDDLEWARE_CLASSES`

Using this approach adds some magicness to your URL routing, because the `AngularUrlMiddleware` class bypasses the HTTP request from normal URL resolving and calls the corresponding view function directly.

## Usage

The reversing functionality is provided by:

```
djangoUrl.reverse(name, args_or_kwargs)
```

This method behaves exactly like Django's URL template tag `{% url 'named:resource' %}`.

### Parameters

- `name`: The URL name you wish to reverse, exactly the same as what you would use in `{% url %}` template tag.
- `args_or_kwargs` (optional): An array of arguments, e.g. `['article', 4]` or an object of keyword arguments, such as `{'type': 'article', 'id': 4}`.

### Examples

A typical Angular Controller would use the service `djangoUrl` such as:

```
var myApp = angular.module('MyApp', ['djng.urls']);

myApp.controller('RemoteItemCtrl', function($scope, $http, $log, djangoUrl) {

  var fetchItemURL = djangoUrl.reverse('namespace:fetch-item');

  $http.get(fetchItemURL).success(function(item) {
    $log.info('Fetched item: ' + item);
  }).error(function(msg) {
    console.error('Unable to fetch item. Reason: ' + msg);
  });
});
```

and with args:

```
$http.get(djangoUrl.reverse('api:articles', [1]))
```

or with kwargs:

```
$http.get(djangoUrl.reverse('api:articles', {'id': 1}))
```

### Parametrized URLs

You can pass a "parametrized" arg or kwarg to `djangoUrl.reverse()` call to be used with $resource.

```
var url = djangoUrl.reverse('orders:order_buyer_detail', {id: ':id'});
// Returns  '/angular/reverse/?djng_url_name=orders%3Aorder_buyer_detail&djng_url_
↪kwarg_id=:id'
// $resource can than replace the ':id' part
```

```
var myRes = $resource(url);
myRes.query({id:2}); // Will call reverse('orders:order_buyer_detail', kwargs={'id':2}
↪) url

// If :param isn't set it will be ignored, e.g.
myRes.query(); // Will call reverse('orders:order_buyer_detail') url

// with @param $resource will autofill param if object has 'param' attribute
var CreditCard = $resource(djangoUrl.reverse('card', {id: ':id'}), {id: '@id'});
var card = CreditCard.get({id: 3}, function (success) {
  card.holder = 'John Doe';
  card.$save() // Will correctly POST to reverse('card', kwargs={'id':3})
})
```

### Additional notes

If you want to override reverse url, e.g. if django app isn't on top level or you want to call another server it can be set in `.config()` stage:

```
myApp.config(function(djangoUrlProvider) {
  djangoUrlProvider.setReverseUrl('custom.com/angular/reverse/');
});
```

**Warning:** The path of request you want to reverse must still remain `/angular/reverse/` on django server, so that middleware knows it should be reversed.

## Resolve Angular Dependencies

As with any application, we also must manage the client side files. They normally are not available from **PyPI** and must be installed by other means than `pip`. This typically is the Node Packet Manager also known as `npm`. When managing a Django project, I strongly recommend to keep external dependencies outside of any asset's folder, such as `static`. They *never shall* be checked into your version control system. Instead change into to the root directory of your project and run

```
npm install npm install <pkg>@<version> --save
```

This command installs third party packages into the folder `node_modules` storing all dependencies inside the file `package.json`. This file shall be added to revision control, whereas we explicitly ignore the folder `node_modules` by adding it to `.gitignore`.

We then can restore our external dependencies at any time by running the command `npm install`. This step has to be integrated into your project's deployment scripts. It is the equivalent to `pip install -r requirements.txt`.

### Accessing External Dependencies

Our external dependencies now are located outside of any static folder. We then have to tell Django where to find them. By using these configuration variables in `settings.py`

```
BASE_DIR = os.path.join(os.path.dirname(__file__)

# Root directory for this Django project (may vary)
PROJECT_ROOT = os.path.abspath(BASE_DIR, os.path.pardir)

STATICFILES_DIRS = (
    os.path.join(BASE_DIR, 'static'),
    ('node_modules', os.path.join(PROJECT_ROOT, 'node_modules')),
)
```

with this additional static files directory, we now can access external assets, such as

```
{% load static %}

<script src="{% static 'node_modules/angular/angular.min.js' %}" type="text/javascript
↪"></script>
```

## Sekizai

django-sekizai is an asset manager for any Django project. It helps the authors of projects to declare their assets in the files where they are required. During the rendering phase, these declared assets are grouped together in central places.

**django-sekizai** is an optional dependency, only required if you want to use the postprocessor.

This helps us to separate concern. We include Stylesheets and JavaScript files only when and where we need them, instead of add every dependency we ever might encounter.

Additionally, in AngularJS we must initialize and optionally configure the loaded modules. Since we do not want to load and initialize every possible AngularJS module we ever might need in any sub-pages of our project, we need a way to initialize and configure them where we need them. This can be achieved with two special Sekizai postprocessors.

### Example

In the base template of our project we initialize AngularJS

Listing 2.1: my_project/base.html

```
{% load static sekizai_tags %}

<html ng-app="myProject">
    <head>
        ...
        {% render_block "css" postprocessor "compressor.contrib.sekizai.compress" %}
        ...
    </head>
    <body>
        ...
```

somewhere in this file, include the minimum set of Stylesheets and JavaScript files required by AngularJS

```
{% addtoblock "js" %}<script src="{% static 'node_modules/angular/angular.min.js' %}"␣
↪type="text/javascript"></script>{% endaddtoblock %}
{% addtoblock "js" %}<script src="{% static 'node_modules/angular-sanitize/angular-
↪sanitize.min.js' %}"></script>{% endaddtoblock %}
```

Before the closing </body>-tag, we then combine those includes and initialize the client side application

```
...
{% render_block "js" postprocessor "compressor.contrib.sekizai.compress" %}
<script type="text/javascript">
angular.module('myProject', ['ngSanitize',
    {% render_block "ng-requires" postprocessor "djng.sekizai_processors.module_list"
↪%}
]).config(['$httpProvider', function($httpProvider) {
    $httpProvider.defaults.headers.common['X-CSRFToken'] = '{{ csrf_token }}';
    $httpProvider.defaults.headers.common['X-Requested-With'] = 'XMLHttpRequest';
}]).config(['$locationProvider', function($locationProvider) {
    $locationProvider.html5Mode(false);
}]){% render_block "ng-config" postprocessor "djng.sekizai_processors.module_config"
↪%};
</script>

</body>
```

Say, in one of the templates which extends our base template, we need the AngularJS animation functionality. Instead of adding this dependency to the base template, and thus to every page of our project, we only add it to the template which requires this functionality.

Listing 2.2: my_project/specialized.html

```
{% extends "my_project/base.html" %}
{% load static sekizai_tags %}

{% block any_inhertited_block_will_do %}
    {% addtoblock "js" %}<script src="{% static 'node_modules/angular-animate/angular-
↪animate.min.js' %}"></script>{% endaddtoblock %}
    {% addtoblock "ng-requires" %}ngAnimate{% endaddtoblock %}
    {% addtoblock "ng-config" %}['$animateProvider', function($animateProvider) {
        // restrict animation to elements with the bi-animate css class with a regexp.
        $animateProvider.classNameFilter(/bi-animate/); }]{% endaddtoblock %}
{% endblock %}
```

Here `addtoblock "js"` adds the inclusion of the additional requirement to our list of external files to load.

The second line, `addtoblock "ng-requires"` adds `ngAnimate` to the list of Angular requirements. In our base template the specified postprocessor `djng.sekizai_processors.module_list` creates a JavaScript array, which is used to initialize our AngularJS application.

The third line, `addtoblock "ng-config"` adds a configuration statement. In our base template this is executed while our AngularJS application configures it's dependencies.

By using these two simple postprocessors inside the templatetag `render_block`, we can delegate the dependency resolution and the configuration of our Angular application to our extended templates. This also applies for HTML snippets included by an extended template.

This approach is a great way to separate concern to the realm it belongs to.

## Three-way data-binding

One of AngularJS biggest selling propositions is its two-way data-binding. Two way data-binding is an automatic way of updating the view whenever the model changes, as well as updating the model whenever the view changes.

With **djng** and the additional module django-websocket-redis, one can extend this feature to reflect all changes to a model, back and forward with a corresponding object stored on the server. This means that the server "sees" whenever

the model on the client changes and can by itself, modify values on the client side at any time, without having the client to poll for new messages. This is very useful, when the server wants to inform the client about asynchronous events such as sport results, chat messages or multi-player game events.

## Installation

If you want to use three-way data-binding with Django, the webbrowser must have support for websockets. Nowadays, most modern browsers do so.

Install **django-websocket-redis** from PyPI:

```
pip install django-websocket-redis
```

and follow the configuration instructions.

## Demo

In the examples directory there is a demo showing the capabilities. If **ws4redis** can be found in the Python search path, this special demo should be available together with the other two examples. Run the demo server:

```
cd examples
./manage runserver
```

point a browser onto http://localhost:8000/threeway_databinding/ and fill the input fields. Point a second browser onto the same URL. The fields content should be the same in all browsers. Change some data, the fields content should update concurrently in all attached browsers.

## Add three-way data-binding to an AngularJS application

Refer to the Javascript file `django-angular.js` somewhere on your page:

```
{% load static %}
<script src="{% static 'djng/js/django-angular.min.js' %}" type="text/javascript"></
↪script>
```

add the module dependency to your application initialization:

```
var my_app = angular.module('myApp', [/* other dependencies */, 'djng.websocket']);
```

configure the websocket module with a URL prefix of your choice:

```
my_app.config(['djangoWebsocketProvider', function(djangoWebsocketProvider) {
    // use WEBSOCKET_URI from django settings as the websocket's prefix
    djangoWebsocketProvider.setURI('{{ WEBSOCKET_URI }}');
    djangoWebsocketProvider.setHeartbeat({{ WS4REDIS_HEARTBEAT }});

    // optionally inform about the connection status in the browser's console
    djangoWebsocketProvider.setLogLevel('debug');
}]);
```

If you want to bind the data model in one of your AngularJS controllers, you must inject the provider **djangoWebsocket** into this controller and then attach the websocket to the server.

```
my_app.controller('MyController', function($scope, djangoWebsocket) {
    djangoWebsocket.connect($scope, 'my_collection', 'foobar', ['subscribe-broadcast',
↪ 'publish-broadcast']);

    // use $scope.my_collection as root object for the data which shall be three-way␣
↪bound
});
```

This creates a websocket attached to the server sides message queue via the module **ws4redis**. It then shallow watches the properties of the object named `'my_collection'`, which contains the model data. It then fires whenever any of the properties change (for arrays, this implies watching the array items; for object maps, this implies watching the properties). If a change is detected, it is propagated up to the server. Changes made to the corresponding object on the server side, are immediately send back to all clients listening on the named facility, referred here as `foobar`.

---

**Note:** This feature is new and experimental, but due to its big potential, it will be regarded as one of the key features in future versions of **django-angular**.

---

# Release History

## 1.1.3

- Fix #309: When using Meta to create forms, `djng.fields.ModelChoiceField` isn't substituted for Django's ModelChoiceField.

## 1.1.2

- Rather than checking if a field in a form that uses the `NgMixins` is in `djng.forms.fields`, check if the field inherits from `DefaultFieldMixin`. Allows the use of custom form fields.

## 1.1.1

- Added spinner to upload areas. Whenever one uploads a file, the wheel spins to improve the client's user experience.

## 1.1

- Instead of adding extra functionality to Django's form fields via inheritance magic, now one must use the corresponding field classes from `djng.forms.fields` if its own form class inheritis from `NgForm` or `NgModelForm`.
- Added support to upload files and images via Ajax.

## 1.0.2

- Added templatetag `djng_locale_script` to include the proper AngularJS locale script.

### 1.0.1

- Fixed #297 "Method `get_context()` on custom Widget is never called": Added class `NgWidgetMixin` which allows to override method `get_context()` in custom widgets.
- Fixed #288 Incorrect `<label for="...">` in widget `RadioChoiceInput`.

### 1.0.0

- Added support for Django 1.10 and 1.11, tests & travis updated.
- Drop support for Django 1.7, 1.8 and 1.9.
- Removed templatetag `{% csrf_value %}`, since Django offers ab equivalent tag.
- Fix file input css (remove the border) and add some documentation about common reported errors.
- Remove support for bower in favor of npm.
- Fix exception while rendering Angular Form using `as_ul`.

### 0.8.4

- Added two optional Sekiazai's postprocessors for better dependency resolution of AngularJS imports and module initialization.

### 0.8.3

- Refactored client side test runner to use npm instead of Grunt.
- Use tox to create the testing matrix.
- Fix #261: ModelMultipleChoiceField and CheckboxSelectMultiple.
- Deprecate `{% csrf_value %}` in favour of `{{ csrf_token }}`.

### 0.8.2

- On the client side, validation of the email address is done using the same regular expression as is used by Django. Until 0.8.1, the less restrictive Angular validation patterns were used.
- Some widgets require more finegrained formatting capabilities.Added a slightly modified method method:*django.forms.utils.flatatt* which can use its own context for formatting.

### 0.8.1

- Fixed a bug in NgModelFormMixin.get_form_data(): Using `and ... or ...` as ternary operator can have unwanted side effects. Also changed other occurrences.

## 0.8.0

- djangular has been renamed to djng and ng.django-... has been renamed to djng-.... This was required by many users since it: - caused a naming conflict with another django app named djangular and - the identifier "djangular" by many users was seen as a bad choice. - violated the AngularJS principle that only their modules shall be prefixed with "ng". Please read https://github.com/jrief/django-angular/issues/35 for the preceded discussion on this topic.
- Support for ngMessages was removed again because - its code base was buggy and unmaintained - it does not make much sense to reduce the amount of auto-generated HTML - it added an alternative form rendering mixin, without any additional functionality
- In the <select> element, the default <option selected="selected"> did not work anymore. This regression was introduced in 0.7.16.

## 0.7.16

- Ready for Django-1.9.
- Fixed: Non-ascii characters were not being processed correctly by django.http.request.QueryDict.__init__.
- In JavaScript, replaced console.log by $log.log.
- Use decimal base on invocation of parseInt.
- Use square brackets to access scope members, which otherwise won't support fields containing −.
- templatetag load_djng_urls has been removed.
- For CRUD, check if request method is allowed.
- Fixed djngError directive, when using AngularJS-1.3.
- Added support for ngMessages, as available with AngularJS-1.3.

## 0.7.15

- Simplified middleware for reversing the URL.
- Reversing url in djangoUrl service can now be overriden.

## 0.7.14

- Supporting Django-1.8.
- The widget bootstrap3.widgets.CheckboxInput got a keyword to set the choice label of a field. This allows to style this kind of field individually in a Django Form.

## 0.7.13

- Change for Forms inheriting from NgFormBaseMixin using field_css_classes as dict: CSS classes specified as default now must explicitly be added the fields defining their own CSS classes. Before this was implicit.
- Added AngularJS directive djng-bind-if. See docs for details.

- Reverted fix for FireFox checkbox change sync issue (135) since it manipulated the DOM. Instead added `scope.$apply()` which fixes the issue on FF.
- In BS3 styling, added `CheckboxFieldRenderer` to `CheckboxInlineFieldRenderer` (the default), so that forms with multiple checkbox input fields can be rendered as block items instead of inlines.
- In BS3 styling, added `RadioFieldRenderer` to `RadioInlineFieldRenderer` (the default), so that forms with multiple radio input fields can be rendered as block items instead of inlines.
- Fixed 'classic form' issue whereby `ngModel` was not being added to `select` of `textarea` elements, so returned errors where not displayed.

## 0.7.12

- No functional changes.

## 0.7.11

- Using `field.html_name` instead of `field.name`. Otherwise `add_prefix()` function on form objects doesn't work properly.
- Fixed Firefox checkbox change sync issue caused by `click` and `change` firing in opposite order to other browsers. Switched to `ng-change` to normalise behaviour.
- Moved rejected error cleanup logic into `field.clearRejected` method, so that it can be removed from anywhere that has access to the field.
- Fixed issue in rejected error clean up loop.
- Added missing subfield cleanup to rejected error cleanup loop.
- Added AngularJS service `djangoUrl` to resolve URLs on the client in the same way as on the server.

## 0.7.10

- Fixed inheritance problem (#122) caused by a metaclass conflicting with Django's `DeclarativeFieldsMetaclass`. This now should fix some issues when using `forms.ModelForm`. This fix changed the API slightly.
- Fixed expansion for templatetag `{% angularjs %}` (#117) when using lists in Python / arrays in JavaScript.

## 0.7.9

- `TupleErrorList` has been adopted to fully support Django-1.7.

## 0.7.8

- Fixed: `ng-minlength` and `ng-maxlength` are not set to `None` if unset.
- Fixed: Concatenated latest version of django-angular.js.

### 0.7.7

- Refactored the code base. It now is much easier to understand the code and to add custom Fields and Widgets.
- Fixed the behaviour of all Widgets offered by Django. They now all validate independently of the method (Post or Ajax) used to submit data to the server.

### 0.7.6

- Fixed regression when using `Bootstrap3FormMixin` in combination with `widgets.CheckboxSelectMultiple`.

### 0.7.5

- Added: Template tag {% angularjs %} which allows to share templates between Django and AngularJS.
- Fixed: Using {{ field.error }} returned unsafe text.
- Fixed: Adjust the regular expression and run grunt build.

### 0.7.4

- Fixed: Error rendering while for hidden input fields.
- Fixed: Bootstrap3 styling: label for field was rendered as lazy object instead of string.
- Added: Url resolvers for angular controllers.

### 0.7.3

- Added support to render a Django Form using a plugable style. Bootstrap3 styling has been implemented.
- Added AngularJS directive for `<input>` fields: They now add a dummy `ngModel` to some input fields, so that Forms using the `NgFormBaseMixin` honor the pristine state and display an error list from the bound form.
- Replaced AngularJS directive for `form` by a directive for `ngModel`. This directive restores the values in bound forms otherwise not vivible in the browser.
- Fixed: Instead of adding attributes to Form Field Widgets, those additional attributes now are added on the fly while rendering. This caused some problems, when Forms were reused in different contexts.
- Fixed: Behavior for BooleanField and MultipleChoiceField has been fixed so AngularJS form validation.

### 0.7.2

- Fixed: select fields, multiple select fields, radio and checkbox input fields and text areas are handled by the built-in form directive to adopt Django's bound forms for AngularJS.

### 0.7.1

- For remote method invocation, replace keyword `action` against a private HTTP-header `DjNg-Remote-Method`. Added template tags `djng_all_rmi` and `djng_current_rmi` which return a list of methods to be used for remote invocation.
- Experimental support for Python-3.3.

### 0.7.0

- Refactored errors handling code for form validation. It now is much easier and more flexible for mixing in other form based classes.
- Added a date validator using an AngularJS directive. * Can be used as a starting point for other customized validators.
- Added another view, which can be used for NgModelMixin without NgValidationMixin.
- Added new directory to handle client code. * Separated JS files for easier development. * Grunt now builds, verifies and concatenates that code. * Karma and Jasmine run unit tests for client code. * A minified version of `django-angular.js` is build by grunt and npm-uglify.
- Rewritten the demo pages to give a good starting point for your own projects.

### 0.6.3

- **ADOPT YOUR SOURCES**: The Javascript file `/static/js/djng-websocket.js` has been moved and renamed to `/static/djangular/js/django-angular.js`
- Internal error messages generated by server side validation, now are mixed with AngularJS's validation errors.
- A special list-item is added to the list of errors. It is shown if the input field contains valid data.
- Input fields of bound forms, now display the content of the field, as expected. This requires the Angular module `ng.django.forms`.

### 0.6.2

- Refactored `NgFormValidationMixin`, so that potential AngularJS errors do not interfere with Django's internal error list. This now allows to use the same form definition for bound and unbound forms.

### 0.6.1

- Bug fix for CRUD view.

### 0.6.0

- Support for basic CRUD view.

### 0.5.0

- Added three way data binding.

### 0.4.0

- Removed @csrf_exempt on dispatch method for Ajax requests.

### 0.3.0

Client side form validation for Django forms using AngularJS

### 0.2.2

- Removed now useless directive 'auto-label'. For backwards compatibility see https://github.com/jrief/angular-shims-placeholder

### 0.2.1

- Set Cache-Control: no-cache for Ajax GET requests.

### 0.2.0

- added handler to mixin class for ajax get requests.
- moved unit tests into testing directory.
- changed request.raw_post_data -> request.body.
- added possibility to pass get and post requests through to inherited view class.

### 0.1.4

- optimized CI process

### 0.1.3

- added first documents

### 0.1.2

- better packaging support

### 0.1.1

- fixed initial data in NgModelFormMixin

### 0.1.0

- initial revision

CHAPTER 3

# Sponsoring

This project is maintained by one individual with help from the community. If it is useful for you, please consider to sponsor it in order to fund its further development. High on the priority list is support for Angular-2 and/or Angular-4.

CHAPTER 4

# Indices and tables

- genindex
- modindex
- search