django-angular Documentation

Release 0.7.14

Jacob Rief

May 08, 2015

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

Project's home

Check for the latest release of this project on Github.

Please report bugs or ask questions using the Issue Tracker.

Contents

2.1 Installation

Install **Django-Angular**. The latest stable release can be found on PyPI

```
pip install django-angular
```

or the newest development version from GitHub

```
pip install -e git+https://github.com/jrief/django-angular#egg=django-angular
```

2.1.1 Dependencies

- [Django](#) >=1.6
- [AngularJS](#) >=1.2

Configuration

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

```
INSTALLED_APPS = (
    ...
    'djangular',
    ...
)
```

Please don't forget to define your `STATIC_ROOT` and `STATIC_URL` properly, then launch the `python manage.py collectstatic` command to update your static content with the JavaScript files provided by **django-angular**.

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

2.2 Integrate AngularJS with Django

2.2.1 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';
});
```

2.2.2 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.

A less clean solution, is to change the syntax of the AngularJS template tags. Just add the following statement during module instantiation:

```
var my_app = angular.module('MyApp').config(function($interpolateProvider) {
    $interpolateProvider.startSymbol('{$');
    $interpolateProvider.endSymbol('$}');
});
```

Now, you can easily distinguish a server side variable substitution `{{ varname }}` from a client side variable substitution `{$ varname $}`.

This approach is less verbose than using the *verbatim* tag. The problem, however, is that you have to remember to use this alternative tag syntax for *all* of your AngularJS templates. It also makes it difficult to integrate 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

---

```python
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`

```python
partial_patterns = patterns('',
    url(r'^partial-template1.html$', PartialGroupView.as_view(template_name='partial-template1.html')
    # ... more partials ...,
)

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

By using the utility function

```python
from djangular.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

```html
<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

```html
<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.

2.2.3 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 'djangular/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 */, 'ng.django.forms']);
```

2.3 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/

2.3.1 Classic Form

Classic Subscribe Form with no data validation.

2.3.2 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.

2.3.3 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.

2.3.4 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.

2.3.5 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.

2.3.6 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.

2.4 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.

2.4.1 Sample code

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

```python
from django import forms
from django.utils import six
from djangular.forms import NgDeclarativeFieldsMetaclass, NgModelFormMixin

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

```python
from djangular.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`:

```python
from djangular.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:

```html
<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`.

2.4.2 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

```python
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`:

```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:

```javascript
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

```python
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

```python
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:

```html
<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.

2.5 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!

2.5.1 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

```python
from django import forms
from django.utils import six
from djangular.forms import NgDeclarativeFieldsMetaclass, NgFormValidationMixin

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

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:

```python
from djangular.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`:

```python
from djangular.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:

```python
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:

```python
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

```html
<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:

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

Note: On Django-1.5, some field constraints, such as the attributes `min_length` and `max_length`, are ignored when used with `NgFormValidationMixin`. This has been fixed in Django-1.6 .

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

```python
from django import forms
from djangular.forms import NgFormValidationMixin

class MyValidatedForm(NgFormValidationMixin, forms.Form):
    email = forms.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.

```html
<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', ['ng.django.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:

```python
from django import forms
from djangular.forms import NgFormValidationMixin, NgModelFormMixin


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

but don't do this

```python
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

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

but this is not

```python
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.

2.5.2 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:

```css
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

```python
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 form validation to customized fields

Django's form validation is not 1:1 compatible with AngularJS's validation. Therefore **djangular** is shipped with a mapping module, which translate Django's form validation to AngularJS. This module is located in `djangular.forms.patched_fields`.

If you need to add or to replace any of these mappings, create a Python module which implements an alternative mapping to the module shipped with **djangular**. Refer to an alternative module in your `settings.py` with the configuration directive `DJANGULAR_VALIDATION_MAPPING_MODULE`.

For further information about how to use form validation with AngularJS, please refer to the *demo pages*.

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:

```python
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})-
```

If your AngularJS application has been initialized with

```
angular.module('MyApp', ['ng.django.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.

2.6 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 **djangular** `NgCRUDView`.

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

2.6.1 Configuration

Subclass `NgCRUDView` and override model attribute:

```
from djangular.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:

```javascript
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.

2.6.2 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.

slug

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.

2.6.3 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.

2.7 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**?

2.7.1 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 djangular.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 djangular_tags %}
...
<script type="text/javascript">
my_app.config(function(djangoRMIProvider) {
    djangoRMIProvider.configure({% djng_all_rmi %});
});
</script>
```

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

Template Tag `djng_current_rmi`

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

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

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

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.

2.7.2 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 = patterns('',
    ...
    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.

2.8 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.

2.8.1 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.

2.9 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.

2.9.1 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.

2.9.2 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:

```python
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 djangular_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 %}
    <img ng-src="{{ item.image.url }}"{% if not ng %} src="{{ item.image.url }}"{% endif %} width="{
    <div{% if ng %} ng-bind-html="item.description"{% endif %}>{% if not ng %}{{ item.description }}
</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.

2.10 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 **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 0.8, **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.

2.10.1 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.

2.10.2 Installation

Angular

- Include `django-angular.js`:

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

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

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

The `djangoUrl` service is now available through [dependency injection](#) to all directives and controllers.

Setting via Django Middleware

- Add `'djangular.middlewares.DjangularUrlMiddleware'` to `MIDDLEWARE_CLASSES` in your Django

`settings.py` file:

```python
MIDDLEWARE_CLASSES = (
    'djangular.middleware.DjangularUrlMiddleware',
    # 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 `DjangularUrlMiddleware` class bypasses the HTTP request from normal URL resolving and calls the corresponding view function directly.

2.10.3 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:

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

myApp.controller('RemoteItemCtrl', ['$scope', '$http', 'djangoUrl', function($scope, $http, djangoUrl

    $scope.loadItem = function() {
        var fetchItemURL = djangoUrl.reverse('namespace:fetch-item');
        $http.get(fetchItemURL).success(function(item) {
            console.log('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}))
```

2.11 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 **djangular** 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.

2.11.1 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.

2.11.2 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.

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

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

```html
<script src="{{ STATIC_URL }}djangular/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 */, 'ng.django.websocket']);
```

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

```
my_app.config(function(djangoWebsocketProvider) {
    // use '/ws' as the websocket's prefix
    djangoWebsocketProvider.prefix('/ws');

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

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.

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

    // 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 the client.

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**.

2.12 Release History

2.12.1 0.7.15-dev

2.12.2 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`.

2.12.3 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.

2.12.4 0.7.12

- No functional changes.

2.12.5 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.

2.12.6 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.

2.12.7 0.7.9

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

2.12.8 0.7.8

- Fixed: `ng-minlength` and `ng-maxlength` are not set to `None` if unset.

- Fixed: Concatenated latest version of django-angular.js.

2.12.9 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.

2.12.10 0.7.6

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

2.12.11 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.

2.12.12 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.

2.12.13 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.

2.12.14 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.

2.12.15 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.

2.12.16 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.

2.12.17 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`.

2.12.18 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.

2.12.19 0.6.1

- Bug fix for CRUD view.

2.12.20 0.6.0

- Support for basic CRUD view.

2.12.21 0.5.0

- Added three way data binding.

2.12.22 0.4.0

- Removed @csrf_exempt on dispatch method for Ajax requests.

2.12.23 0.3.0

Client side form validation for Django forms using AngularJS

2.12.24 0.2.2

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

2.12.25 0.2.1

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

2.12.26 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.

2.12.27 0.1.4

- optimized CI process

2.12.28 0.1.3

- added first documents

2.12.29 0.1.2

- better packaging support

2.12.30 0.1.1

- fixed initial data in NgModelFormMixin

2.12.31 0.1.0

- initial revision

Indices and tables

- genindex
- modindex
- search