# editlive Documentation

***Release 0.1.0***

**Maxime Haineault**

**Sep 27, 2017**

# Contents

**Author** Maxime Haineault <max@motion-m.ca>

**Source code** github.com project

**Bug tracker** github.com issues

**Generated** Sep 27, 2017



**Version** 0.1.0

## Everything you need to know about Django editlive.

**Live examples**

You can see editlive in action on here

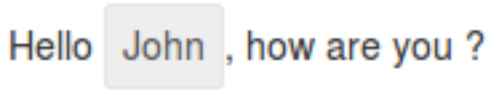
Django editlive is a Free Open Source project which aims to make it easy to make elegant inline editable fields from database objects.

Here's a simple example:

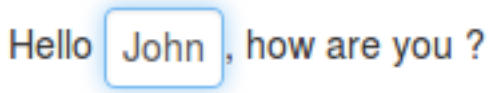
```
{% load editlive_tags %}

{% editlive "request.user.first_name" as firstname %}
Hello {{ firstname }}, how are you ?
```

This would output something like *Hello [John], how are you ?*.




The name *[John]* would be a clickable *span* element called the "placeholder".




When the placeholder is clicked, it's replaced by an input field and when this field is blurred, it is automatically saved to the database with AJAX. To cancel an edit you must use the escape key.

Adaptors are special classes which are used to abstract the different fields types and interact with them. Learn more about it below:

CHAPTER 1

# Table of Contents

## Installation

First you will need to install Dajaxice. If you use pip or buildout, simply add it to your requirements and rebuild your environment.

Then add *dajaxice.finders.DajaxiceFinder* to your *settings.STATICFILES_FINDERS*.

Add the Dajaxice JS to your base template:

```
{%load dajaxice_templatetags%}
{%dajaxice_js_import%}
```

Add the editlive CSS and JS to your base template:

```
{% load dajaxice_templatetags %}
{# jQuery + jQuery UI + Bootstrap #}
<link rel="stylesheet" type="text/css" href="{{ STATIC_URL }}editlive/contrib/jquery-
↪ui-bootstrap/css/style.css">
<link rel="stylesheet" type="text/css" href="{{ STATIC_URL }}editlive/contrib/
↪bootstrap/css/bootstrap-responsive.min.css">
<script type="text/javascript" src="{{ STATIC_URL }}editlive/contrib/jquery-ui-
↪bootstrap/js/jquery.min.js"></script>
<script type="text/javascript" src="{{ STATIC_URL }}editlive/contrib/jquery-ui-
↪bootstrap/js/jquery-ui.custom.min.js"></script>
<script type="text/javascript" src="{{ STATIC_URL }}editlive/contrib/jquery-ui-
↪bootstrap/js/bootstrap/bootstrap.min.js"></script>
<link rel="stylesheet" type="text/css" href="{{ STATIC_URL }}editlive/contrib/jquery-
↪ui-bootstrap/third-party/jQuery-UI-Date-Time-Picker/jquery-ui-timepicker.css">
<script type="text/javascript" src="{{ STATIC_URL }}editlive/contrib/jquery-ui-
↪bootstrap/third-party/jQuery-UI-Date-Time-Picker/jquery-ui-timepicker-addon.js"></
↪script>
<script type="text/javascript" src="{{ STATIC_URL }}editlive/contrib/bootstrap/js/
↪bootstrap.min.js"></script>
{% dajaxice_js_import %}
```

```
{# Editlive #}
<link rel="stylesheet" type="text/css" href="{{STATIC_URL}}editlive/css/editlive.css">
<script src="{{STATIC_URL}}editlive/js/jquery.editlive.js"></script>
<script src="{{STATIC_URL}}editlive/js/jquery.editlive.char.js"></script>
<script src="{{STATIC_URL}}editlive/js/jquery.editlive.text.js"></script>
<script src="{{STATIC_URL}}editlive/js/jquery.editlive.date.js"></script>
<script src="{{STATIC_URL}}editlive/js/jquery.editlive.datetime.js"></script>
<script src="{{STATIC_URL}}editlive/js/jquery.editlive.boolean.js"></script>
<script src="{{STATIC_URL}}editlive/js/jquery.editlive.foreignkey.js"></script>
<script src="{{STATIC_URL}}editlive/js/jquery.editlive.choices.js"></script>
<script src="{{STATIC_URL}}editlive/js/jquery.editlive.manytomany.js"></script>
```

**Note:** not all files are required. Eventually all the JS will be minified. Meanwhile you can include what you need or use something like django-pipeline.

**Protip:** You can add the following logger to your settings.LOGGING['loggers'] to redirect dajaxice exceptions to the console.

```
'dajaxice': {
    'handlers': ['console'],
    'level': 'WARNING',
    'propagate': False,
},
```

# Settings

## EDITLIVE_DATE_WIDGET_FORMAT

This setting is used to pass the date format to the datepicker widget.

The format used must match one of the format of the django setting *DATE_INPUT_FORMAT* which itself is used by django to parse and validate input content.

By default *DATE_INPUT_FORMAT* is set to the following:

```
('%Y-%m-%d %H:%M:%S', '%Y-%m-%d %H:%M', '%Y-%m-%d',
'%m/%d/%Y %H:%M:%S', '%m/%d/%Y %H:%M', '%m/%d/%Y',
'%m/%d/%y %H:%M:%S', '%m/%d/%y %H:%M', '%m/%d/%y')
```

Here's a translation table for Python / Django / jQuery UI date format:

| Py | Dj | Js | Description |
|---|---|---|---|
| | j | d | day of month (no leading zero) |
| d | d | dd | day of month (two digit) |
| | z | o | day of the year (no leading zeros) |
| j | z | oo | day of the year (three digit) * |
| a | D | D | day name short |
| A | l | DD | day name long |
| | n | m | month of year (no leading zero) |
| m | m | mm | month of year (two digit) |
| b | M | M | month name short |
| B | F | MM | month name long |
| y | y | y | year (two digit) |
| Y | Y | yy | year (four digit) |
| | U | @ | Unix timestamp (ms since 01/01/1970) |

As you can see .. this is quite a mess:

- Django use the setting *DATE_FORMAT* to render dates in template. It has its own date formatting implementation described in the builtin date template filter documentation.
- Then to parse and validate date inputs it uses the *DATE_INPUT_FORMAT* which uses the Python strptime format
- And finally, the *DATE_WIDGET_FORMAT* is used by editlive to set the datepicker format, which must validate against a *DATE_INPUT_FORMAT*. The *DATE_WIDGET_FORMAT* use the jQuery UI date format as described in the datepicker documentation.

At this point you might want to take a little time for yourself and cry a little bit.

## EDITLIVE_TIME_WIDGET_FORMAT

This setting is used to pass the time format to the timepicker plugin.

In Django the datetime format is specified as a single argument (for example: *%Y-%m-%d %H:%M:%S*).

But jQuery UI uses two seperate settings for the date and time formats.

Here's the translation table for the time formatting:

| Py | Dj | Js | Description |
|---|---|---|---|
| H | h-H | hh | Hour, 12/24-hour format. |
| | g | h | Hour, 12/24-hour format without zeros |
| M | i | mi | Minutes with zeros. |
| | | m | Minutes (unsupported by django) |
| S | s | ss | Seconds, 2 digits with leading zeros |
| | | s | Seconds (unsupported by django) |
| f | u | l | Microseconds |
| Z | T | z | Time zone |
| | | t | AM/PM (unsupported by django) |
| P | A | tt | AM/PM |

You can find the full timepicker formatting reference here.

## EDITLIVE_ADAPTORS

This setting serves as default mapping between field types and adaptors.

Currently not all field types are supported, here's the current default mapping:

```
EDITLIVE_DEFAULT_ADAPTORS = {
    'char':     'editlive.adaptors.CharAdaptor',
    'text':     'editlive.adaptors.TextAdaptor',
    'date':     'editlive.adaptors.DateAdaptor',
    'datetime': 'editlive.adaptors.DateTimeAdaptor',
    'time':     'editlive.adaptors.TimeAdaptor',
    'boolean':  'editlive.adaptors.BooleanAdaptor',
    'fk':       'editlive.adaptors.ForeignKeyAdaptor',
    'choices':  'editlive.adaptors.ChoicesAdaptor',
    'm2m':      'editlive.adaptors.ManyToManyAdaptor',
}
```

If you want to override the datetime adaptor with your own, you can simply provide one in your *settings.py* like so:

```
EDITLIVE_ADAPTORS = {
    'datetime': 'mymodule.adaptors.MyDateTimeAdaptor',
}
```

The settings *EDITLIVE_ADAPTORS* updates the adaptor mapping instead of overwriting it, so the end result would be this:

```
EDITLIVE_DEFAULT_ADAPTORS = {
    'char':     'editlive.adaptors.CharAdaptor',
    'text':     'editlive.adaptors.TextAdaptor',
    'date':     'editlive.adaptors.DateAdaptor',
    'datetime': 'mymodule.adaptors.MyDateTimeAdaptor',
    'time':     'editlive.adaptors.TimeAdaptor',
    'boolean':  'editlive.adaptors.BooleanAdaptor',
    'fk':       'editlive.adaptors.ForeignKeyAdaptor',
    'choices':  'editlive.adaptors.ChoicesAdaptor',
    'm2m':      'editlive.adaptors.ManyToManyAdaptor',
}
```

# Usage examples

## Basic usage

In a template, editlive can take any in context database object and make it editable live with a simple template tag:

```
{% load editlive_tags %}

{% editlive "object.description" as object_description %}
<div>
    {{ object_description }}
</div>
```

This will render the object's property value in a clickable container. When the container is clicked, it changes to a input field according to the field's type.

It's possible to apply template filters to the placeholder's display value like this:

```
{% editlive "object.description" template_filters="capfirst" as object_description %}

{% editlive "object.date_visit" template_filters="date:'l j M Y at H:i\h'" as date_
↪visit %}
```

Most other arguments are converted into js options and fed to the jQuery UI widget.

## Working with formsets

Formsets are a bit tricky since you need to edit multiple fields with the same id and name attributes.

So for this to work, the id and name attributes must be altered to make them unique. To achieve this, simply pass a formset argument to editlive and give it a meaningful name:

```
{% editlive "object.first_name" formset="user" as user_firstname %}
{{ user_firstname }}
```

The input field will then look like this:

```
<input type="text" maxlength="250" name="user_set-0-first_name" id="id_user_set-0-
↪first_name" />
```

### How it works

To avoid conflicting with other plugins or altering the input field directly, editlive use its own tag to bind the field properties and settings to the right input.

For example, if we were to *editlive* the *first_name* property of a user object, the output would look something like this:

```
<input type="text" maxlength="250" value="Bob" name="first_name" id="id_first_name" />
<editlive app-label="auth" module-name="user" field-name="first_name" data-field-id=
↪"id_first_name" data-type="textField" object-id="1" rendered-value="Bob" />
```

This way *editlive* stays non-intrusive as it doesn't alter the original input tag.

This also means that you are not constrained to use the editlive template tag, you can hardcode *<editlive />* tag in HTML and the JavaScript will hook it up.

## Options

The editlive template tags accepts three kind of options.

- **template tag options**: these options affect the template tag behavior
- **data options**: options starting with *data_*
- **widget options**: all other options you may pass will be sent to the jQuery UI widget

### Template tag options

#### wrapclass

Add a CSS class to the control's container.:

```
{%load editlive_tags%}
{%editlive "object.date_test" wrapclass="lead" as field%}{{field}}
```

#### load_tags

Editlive fields are rendered with standard django template. This option is used to load extra template tags in these template instance. As is this option is not really useful, it's more a complement to the next option.

#### template_filters

With the *template_filters* option you can control the rendering of the placeholder value.

### mini

Smaller placeholder/input.

### maxwidth

Set max width of the placeholder.

### width

Set the width of the placeholder.

### readonly

Toggle readonly mode:

```
{%load editlive_tags%}
{%editlive "object.date_test" readonly=object.is_archived as field%}{{field}}
```

### formset

If you are iterating over a object set you will need to use the formset argument so each field as its own id.

```
<table>
    {% for line in object.relatedobject_set.all %}
    <tr>
        <td>{% editlive "line.name" formset="myformset" line_name %}{{ line_name }}</
↪td>
        <td>{% editlive "line.email" formset="myformset" line_email %}{{ line_email }}
↪</td>
    </tr>
    {% endfor %}
</table>
```

### class

Add class to the wrapper. A *fixedwidth* class helper is provided. When used in conjunction with *width*, if the text of the placeholder is wider than its container will be truncated and an elipsis will be added:

class="fixedwidth span1"

```
{%load editlive_tags%}
{%editlive "object.date_test" width="100" class="fixedwidth" as field%}{{field}}
```

# The JavaScript API

## Events

Editlive default widgets provides a set of events which follow the jQuery UI model.

If your custom widget subclass the charField widget like the default widgets, these events will also be available. Just remember to trigger them if you override a method.

### blur

Triggered when a field is blurred (once the placeholder is shown)

```
$(function(){
    $('#id_field').on('editliveblur', function(event, ui){
        var element = $(this); // Input
        var editliveInstance = ui;
        // Do something
    });
});
```

### change

Last event triggered after a succesfull save.

```
$(function(){
    $('#id_field').on('editlivechange', function(event, ui){
        var element = $(this); // Input
        var editliveInstance = ui;
        // Do something
    });
});
```

### error

Triggered when a validation error occurs.

```
$(function(){
    $('#id_field').on('editliveerror', function(event, ui){
        var element = $(this); // Input
        var editliveInstance = ui;
        // Do something
    });
});
```

### focus

Triggered before showing the field.

```
$(function(){
    $('#id_field').on('editlivefocus', function(event, ui){
        var element = $(this); // Input
        var editliveInstance = ui;
        // Do something
    });
});
```

### focused

Triggered once the field is shown.

```
$(function(){
    $('#id_field').on('editlivefocus', function(event, ui){
        var element = $(this); // Input
        var editliveInstance = ui;
        // Do something
    });
});
```

### save

Triggered before saving.

```
$(function(){
    $('#id_field').on('editlivesave', function(event, ui){
        var element = $(this); // Input
        var editliveInstance = ui;
        // Do something
    });
});
```

### success

Triggered after a succesfull save.

```
$(function(){
    $('#id_field').on('editlivesuccess', function(event, ui){
        var element = $(this); // Input
        var editliveInstance = ui;
        // Do something
    });
});
```

## $.fn.editlive

This is a gateway function used to interact with an instancied editlive field.

Val is used to get and set the field value:

```
$('#id_char_test').editlive('val');
"Hello World"

$('#id_char_test').editlive('val', 'Hello Universe');
"Hello Universe"
```

## $.editlive.load

Scan the entire document for editlive tags and load their widgets:

```
$(function(){
    $.editlive.load();
});
```

This is equivalent of doing this:

```
$(function(){
    $('editlive').each(function(k, v) {
        $.editlive.loadWidget(v);
    });
});
```

You can also pass a selector as parent:

```
$(function(){
    $.editlive.load('#my-ajax-content-wrapper');
});
```

### $.editlive.loadWidget

Load a given editlive widget element:

```
$(function(){
    var widget = $('editlive:first');
    $.editlive.loadWidget(widget);
});
```

## Adaptors

Adaptor are special class which are used as abstract API to work with editlive objects. They provide basic functionalities such as rendering, validating and updating an object. Each django field types can have its own adaptor.

Currently, the following adaptors are provided as default:

### editlive.adaptors.base – Base Adaptor

**class** editlive.adaptors.base.**BaseAdaptor**(*request*, *field*, *obj*, *field_name*, *field_value=''*, *kwargs={}*)

Bases: object

The BaseAdaptor is an abstract class which provides all the basic methods and variable necessary to interact with and render an object field.

It provides the following functionalities:

- Renders the object's display value (rendered with filters applied)
- Renders the editlive markup
- Get a field value
- Set a field value
- Validate field value
- Save a field value

**kwargs**

---

**Note:** This class is never used directly, you must subclass it.

---

**can_edit**()

**format_attributes**()
Formats the HTML attributes of the <editlive /> element.

This method takes no argument, it only use *self.kwargs* to build up *self.attributes* then convert it to HTML attributes and return it as a string.

```
>>> self.format_attributes()
app-label="myapp" field-name="myfieldname"
rendered-value="Hello World" object-id="1"
data-type="charField" data-field-id="id_myfieldname"
module-name="mymodule"
```

Most kwargs are prefixed with *data-* when converted to attribute except for those:

- rendered-value
- load_tags

**get_form**()
Creates and returns a form on the fly from the model using *modelform_factory*.

The returned form is instantiated with *self.obj*.

**get_real_field_name**()
Returns the reald fieldname regardless if the field is part of a formset or not.

Formsets mangles fieldnames with positional slugs. This method returns the actual field name without the position.

```
>>> print self.field_name
"myfieldname_set-0"
>>> print self.get_real_field_name()
"myfieldname"
```

**get_value**()
Returns *self.field_value* unless it is callable. If it is callable, it calls it before returning the output.

**render**()
Returns the form field along with the <editlive /> tag as string

```
>>> self.render()
<input id="id_firstname" type="text" name="firstname" maxlength="25" />
<editlive app-label="myapp" field-name="firstname"
    rendered-value="John" object-id="1" data-type="charField"
    data-field-id="id_firstname" module-name="mymodule"></editlive>
```

**render_value**(*value=None*)
Returns the field value as it should be rendered on the placeholder.

**render_widget**()
Returns the <editlive /> HTML widget as string.

This will also set the *self.attributes['rendered-value']*.

```
>>> self.render_widget()
<editlive app-label="myapp" field-name="firstname"
    rendered-value="John" object-id="1" data-type="charField"
    data-field-id="id_firstname" module-name="mymodule"></editlive>
```

**save**()
Saves the object to database.

A form is created on the fly for validation purpose, but only the saved field is validated.

**Successful save**

```
>>> self.set_value('john@doe.com')
'john@doe.com'
>>> self.save()
{'rendered_value': u'john@doe.com', 'error': False}
```

**Validation error**

```
>>> self.set_value('Hello world')
'Hello world'
>>> self.save()
{'rendered_value': u'Hello world', 'error': True, 'value': u'Hello world',
 'messages': [{'message': u'Enter a valid e-mail address.', 'field_name':
↪'email_test'}]}
```

**set_value**(*value*)
Set the value of the object (but does not save it) and sets *self.field_value*.

## editlive.adaptors.boolean – Boolean Adaptor

**class** editlive.adaptors.boolean.**BooleanAdaptor**(**args*, ***kwargs*)
Bases: *editlive.adaptors.base.BaseAdaptor*

The BooleanAdaptor is used for BooleanField.

---

**Note:** Not tested with NullBooleanField.

---

**get_value**()
Instead of returning True or False we return 'on' or 'off'

## editlive.adaptors.char – Char Adaptor

**class** editlive.adaptors.char.**CharAdaptor**(**args*, ***kwargs*)
Bases: *editlive.adaptors.base.BaseAdaptor*

The CharAdaptor is used for CharField and unknown field types".

## editlive.adaptors.choices – Choices Adaptor

**class** editlive.adaptors.choices.**ChoicesAdaptor**(**args*, ***kwargs*)
Bases: *editlive.adaptors.base.BaseAdaptor*

The ChoicesAdaptor is used for fields with a *choices* argument.

**get_value**()
Instead of returning the field value we call and return the object's *get_FIELD_display* method.

## editlive.adaptors.boolean – Date and DateTime Adaptors

**class** editlive.adaptors.date.**DateAdaptor**(**args*, ***kwargs*)
Bases: *editlive.adaptors.base.BaseAdaptor*

DateField adaptor

Uses the following setting:

*settings.EDITLIVE_DATE_FORMAT*

**render_value**(*value=None*)

**class** editlive.adaptors.date.**DateTimeAdaptor**(**args*, ***kwargs*)
Bases: *editlive.adaptors.base.BaseAdaptor*

DateTimeField adaptor

Uses the following setting:

*settings.EDITLIVE_DATE_FORMAT settings.EDITLIVE_TIME_FORMAT*

**render_value**(*value=None*)

**class** editlive.adaptors.date.**TimeAdaptor**(**args*, ***kwargs*)
Bases: *editlive.adaptors.base.BaseAdaptor*

TimeField adaptor

Uses the following setting:

*settings.EDITLIVE_TIME_FORMAT*

**render_value**(*value=None*)

## editlive.adaptors.foreignkey – ForeignKey Adaptor

**class** editlive.adaptors.foreignkey.**ForeignKeyAdaptor**(**args*, ***kwargs*)
Bases: *editlive.adaptors.base.BaseAdaptor*

The ForeignKeyAdaptor is used for ForeignKey fields".

**render_value**(*value=None*)

**set_value**(*value*)

## editlive.adaptors.manytomany – ManyToMany Adaptor

**class** editlive.adaptors.manytomany.**ManyToManyAdaptor**(**args*, ***kwargs*)
Bases: *editlive.adaptors.base.BaseAdaptor*

The ManyToManyAdaptor is used for ManyToMany fields".

**set_value**(*value*)

### editlive.adaptors.text – Text Adaptor

**class** editlive.adaptors.text.**TextAdaptor**(**args*, ***kwargs*)
Bases: *editlive.adaptors.base.BaseAdaptor*

The TextAdaptor is used for Text fields.

## UI widgets

UI widgets are basically just Jquery UI widget which respect a certain API.

### JsDoc Reference

#### _global_

#### Methods

**_renderItem**(*ul*, *item*)

**Arguments**

- **ul** –
- **item** –

#### jQuery

See (http://jquery.com/).

#### Constructor

**class jQuery**()

#### jQuery.fn

See (http://jquery.com/)

#### Constructor

**class** jQuery.**fn**()

#### jQuery.fn.booleanField

booleanField - Widget for boolean fields (drop down menu)

#### Constructor

**class** jQuery.fn.**booleanField**()

### jQuery.fn.charField

charField - the base widget

### Constructor

**class** `jQuery.fn.`**`charField`**`()`

### Methods

`charField.`**`_set_value`**`(`*`v`*`)`

**Arguments**

- **v** (*mixed*) –
  - The new value.

**Returns** v - The new value.

**Return type** mixed

Updates the internal widget value and the DOM element's value

### jQuery.fn.choicesField

choicesField - A widget for choices fields (dropdown menu)

### Constructor

**class** `jQuery.fn.`**`choicesField`**`()`

### jQuery.fn.dateField

dateField - A date picker widget

### Constructor

**class** `jQuery.fn.`**`dateField`**`()`

### jQuery.fn.datetimeField

datetimeField - A datetime picker widget

### Constructor

**class** `jQuery.fn.`**`datetimeField`**`()`

#### jQuery.fn.foreignkeyField

foreignkeyField - Autocomplete widget for foreignkey field

#### Constructor

**class** `jQuery.fn.`**`foreignkeyField`**`()`

#### jQuery.fn.foreignkeyFieldSelect

foreignkeyFieldSelect - Standard dropdown select for foreignkey field

#### Constructor

**class** `jQuery.fn.`**`foreignkeyFieldSelect`**`()`

#### jQuery.fn.manytomanyField

manytomanyField - Widget for many to many field

#### Constructor

**class** `jQuery.fn.`**`manytomanyField`**`()`

#### jQuery.fn.textField

textField - Widget for text field (textarea)

#### Constructor

**class** `jQuery.fn.`**`textField`**`()`

## Creating and using custom UI widgets

### Extending existing widgets

Sometimes you need to use a custom widget for a field, or you simply want to use another UI library.

Here's an example which use a dropdown menu instead of a switch button for a boolean field:

```
;(function($){

    var booleanDropDown = {
        _type: 'boolean',
        options: {
            choices: 'Yes|No'
        }
    };
```

```
    booleanDropDown._init = function() {
        var $self = this,
            label = $self.options.choices.split('|')

        $self.label_on = label[0];
        $self.label_off = label[1];
        $self.element.hide();
        $self.btn_group = $('<div class="btn-group" />').insertAfter($self.element);
        $self.btn_label = $('<button class="btn" />').appendTo($self.btn_group);
        $self.choices   = $('<ul class="dropdown-menu" />').appendTo($self.btn_group);
        $self.btn_toggle = $(['<button class="btn dropdown-toggle" data-toggle=
↪"dropdown">',
                              '<span class="caret"></span></button>'].join(''))
                              .insertAfter($self.btn_label);
        $self._populate();
    };

    booleanDropDown._populate = function() {
        var $self = this;
        if ($self.element.is(':checked')) {
            $('<li><a href="#off">'+ $self.label_off +'</a></li>')
                .appendTo($self.choices)
                .bind('click.editlive', function(){
                    $self.btn_group.removeClass('open');
                    $self.choices.find('li').remove();
                    $self.element.prop('checked', false);
                    $self.change();
                    $self._populate();
                    return false;
                });
        }
        else {
            $('<li><a href="#on">'+ $self.label_on +'</a></li>')
                .appendTo($self.choices)
                .bind('click.editlive', function(){
                    $self.btn_group.removeClass('open');
                    $self.choices.find('li').remove();
                    $self.element.prop('checked', true);
                    $self.change();
                    $self._populate();
                    return false;
                });
        }
        $self.btn_label.text(this.get_value_display());
    };

    booleanDropDown._display_errors = function(errors) {
        var $self = this;
        $.each(errors, function(k, v) {
            var el = $self.btn_group;
            el.tooltip({
                title: v.message,
                placement: $self.options.errorplacement
            }).tooltip('show');
        });
    };
```

```
    booleanDropDown.get_value_display = function() {
        var $self = this;
        if ($self.element.is(':checked')) {
            return $self.label_on;
        }
        else {
            return $self.label_off;
        }
    };

    booleanDropDown._get_value = function() {
        if (this.element.is(':checked')) return true;
        else return false;
    };

    $.widget('editliveWidgets.booleanDropDown', $.editliveWidgets.charField,␣
↪booleanDropDown);

})(jQuery);
```

### Create a widget from scratch

#### The JavaScript

Barebone example:

```
;(function($){

    $.widget('editliveWidgets.autocompleteField', $.editliveWidgets.charField, {
        _type: 'autocomplete',
        options: {}
    });

})(jQuery);
```

As is, this plugin will act exactly as a *char* field.

The charField is the base field widget, which means that looking at the source code of $.editliveWidgets.charField basically defines the editlive widget API.

For example, let's say we want to activate a autocomplete plugin on our field.

We'd simply override the *_create* method like this:

```
;(function($){

    $.widget('editliveWidgets.autocompleteField', $.editliveWidgets.charField, {
        _type: 'autocomplete',
        options: {},

        _create: function() {
            var $self = this;
            $.editliveWidgets.charField.prototype._create.apply(this, arguments);
            $self.element.autocomplete({minKeys: 2});
        }
    });
```

```
})(jQuery);
```

For more examples please refer to the widgets source code.

### The adaptor

In order to use your custom widget you will have to create a custom adaptor.

Fortunately, this is quite trivial. Here's the date picker adaptor for example:

```
class DateAdaptor(BaseAdaptor):
    """DateField adaptor"""

    def __init__(self, *args, **kwargs):
        """
        The DateAdaptor override the __init__ method
        to add the data-format argument to the
        widget's attributes. This is what links Django internal
        date format with the UI.
        """
        super(DateAdaptor, self).__init__(*args, **kwargs)
        field = self.form.fields.get(self.field_name)
        if field:
            self.attributes.update({'data-format': '%s' % field.widget.format})
        if self.form_field:
            self.attributes.update({'data-type': 'dateField'})

    def render_value(self, value=None):
        """
        If no custom "date" template filter is passed as argument,
        we add one using the settings.DATE_FORMAT as value.
        """
        if self.template_filters is None:
            self.template_filters = []
        if not any(i.startswith('date:') for i in self.template_filters):
            self.template_filters.append(u"date:'%s'" % settings.DATE_FORMAT)
        return unicode(super(DateAdaptor, self).render_value(value=value))
```

You can browse the adaptors source code for more examples.

### Using a custom adaptor

To use a custom adaptor, simply pass a *widget* argument to editlive:

```
{% editlive "object.date_start" widget="mymodule.adaptors.MyCustomDatePickerAdaptor"␣
↪as field %}
{{ field }}
```

# Developers

On this page you will find information for developers who want to contribute to this project.

### editlive.utils – Utils

Editlive utils provide utility functions to perform editlive related tasks.

editlive.utils.**apply_filters**(*value*, *filters*, *load_tags=None*)

editlive.utils.**encodeURI**(*uri*)
We really only need to escape ” (double quotes) and non-ascii characters..

editlive.utils.**get_adaptor**(*request*, *obj*, *field_name*, *field_value=None*, *kwargs={}*, *adaptor=None*)

editlive.utils.**get_default_adaptor**(*field*)

editlive.utils.**get_dict_from_obj**(*obj*)

editlive.utils.**get_dynamic_modelform**(***kwargs*)

editlive.utils.**get_field_type**(*field*)

editlive.utils.**import_class**(*classpath*, *package=None*)

editlive.utils.**import_module**(*name*, *package=None*)

editlive.utils.**is_managed_field**(*obj*, *fieldname*)

editlive.utils.**isinstanceof**(*field*, *types*)

## Testing and development environment

An example project is included to provide quick bootstraping of a development and testing environment.

### Create a virtualenv

```
cd django-editlive/
virtualenv --distribute --no-site-packages editlive_test_env
source editlive_test_env/bin/activate
```

### Install requirements

```
pip install -r docs/requirements.txt
pip install -r example_project/requirements.txt
```

### Install Google Chrome & Google Chrome Webdriver

**Note:** This installation has only been tested on Ubuntu 12+.

```
# Install Google Chrome (if not already installed!)
wget https://dl.google.com/linux/direct/google-chrome-stable_current_i386.deb
sudo apt-get install libgconf2-4
sudo dpkg -i google-chrome*.deb

# Install the Google Chrome webdriver
wget https://chromedriver.googlecode.com/files/chromedriver_linux32_23.0.1240.0.zip
unzip chromedriver_linux32_23.0.1240.0.zip
mv chromedriver /usr/local/bin
```

## Contributing to documentation

You see that *build* script in the docs folder ? Don't use it.

That is, unless you have followed the instructions on how to compile the JavaScript documentation and placed the *sdoc_toolkit-2.4.0* in a folder named *~/UbuntuOne/SDKs*.

I might give the build script some attention some day and make it more useful, but for now I have other priorities.

### Writing documentation

You can find documentation in three places in this project:

- In Python docstring
- In JavaScript comments of the editlive widgets
- And finally in the *docs/* folder

So if you submit a pull request, it's quite easy to update the documentation for the code you are submitting. You just have to comment the code properly.

### Building the Python documentation

```
cd django-editlive/docs/
make html
```

### Building the JavaScript documentation

This is only needed if changes have been made to a JavaScript file.

### Installing requirements

Using Ubuntu One is really not a requirement, just a convenience for me.

```
mkdir -p ~/Ubuntu\ One/SDKs/ && cd ~/Ubuntu\ One/SDKs/
wget http://jsdoc-toolkit.googlecode.com/files/jsdoc_toolkit-2.4.0.zip
unzip jsdoc_toolkit-2.4.0.zip
cd jsdoc_toolkit-2.4.0
```

### Compiling docs

```
cd django-editlive/
java -jar ~/Ubuntu\ One/SDKs/jsdoc_toolkit-2.4.0/jsdoc-toolkit/jsrun.jar \
~/Ubuntu\ One/SDKs/jsdoc_toolkit-2.4.0/jsdoc-toolkit/app/run.js ./ \
--template=_themes/jsdoc-for-sphinx -x=js,jsx --directory=./jsdoc
```

### Including documentation

```
.. include:: jsdoc/MyJavascriptClass.rst
   :start-after: class-methods
```

### Tags reference

- **@augments** - Indicate this class uses another class as its "base."
- **@author** - Indicate the author of the code being documented.
- **@argument** - Deprecated synonym for **@param**.
- **@borrows** that as this - Document that class's member as if it were a member of this class.
- **@class** - Provide a description of the class (versus the constructor).
- **@constant** - Indicate that a variable's value is a constant.
- **@constructor** - Identify a function is a constructor.
- **@constructs** - Identicate that a lent function will be used as a constructor.
- **@default** - Describe the default value of a variable.
- **@deprecated** - Indicate use of a variable is no longer supported.
- **@description** - Provide a description (synonym for an untagged first-line).
- **@event** - Describe an event handled by a class.
- **@example** - Provide a small code example, illustrating usage.
- **@extends** - Synonym for **@augments**.
- **@field** - Indicate that the variable refers to a non-function.
- **@fileOverview** - Provides information about the entire file.
- **@function** - Indicate that the variable refers to a function.
- **@ignore** - Indicate JsDoc Toolkit should ignore the variable.
- **@inner** - Indicate that the variable refers to an inner function (and so is also **@private**).
- **@lends** - Document that all an object literal's members are members of a given class.
- {**@link** ...} - Like **@see** but can be used within the text of other tags.
- **@memberOf** - Document that this variable refers to a member of a given class.
- **@name** - Force JsDoc Toolkit to ignore the surrounding code and use the given variable name instead.
- **@namespace** - Document an object literal is being used as a "namespace."
- **@param** - Describe a function's parameter.
- **@private** - Indicate a variable is private (use the -p command line option to include these).
- **@property** - Document a property of a class from within the constructor's doclet.
- **@public** - Indicate an inner variable is public.
- **@requires** - Describe a required resource.
- **@returns** - Describe the return value of a function.
- **@see** - Describe a related resource.

- **@since** - Indicate that a feature has only been available on and after a certain version number.
- **@static** - Indicate that accessing the variable does not require instantiation of its parent.
- **@throws** - Describe the exception that a function might throw.
- **@type** - Describe the expected type of a variable's value or the value returned by a function.
- **@version** - Indicate the release version of this code.

### References

- Sphinx - Python documentation generator
- Sphinx - Include documentation from docstrings
- JSDOC - A documentation generator for JavaScript
- JSDOC for Sphinx

## Behavior Driven Testing

**Tip for committers**

If you send pull request to this project for things that does not requires testing, like updating the documentation or fixing typos in comments, just add *[ci skip]* in your commit message and a build wont be triggered on Travis CI.

As editlive is a Interface component, a BTD approach is used for testing it.

This gives this project the following benefits:

- Writing test is *really* easy. Non-programmers could almost write them.
- It tests the client and the backend at the same time, like a real user would
- It allow real world testing in multiple browsers (but currently only in Firefox and Chrome are tested)

Travis CI is used as build server. Not only you can see the current build status and the complete history, but you can see the build status of branches and pull requests.

The test suite is a mix of Lettuce, Selenium and Splinter.

### Lettuce

Lettuce makes writing the test cases and scenarios very easy

### Features

```
Feature: Manipulate strings
  In order to have some fun
  As a programming beginner
  I want to manipulate strings

  Scenario: Uppercased strings
    Given I have the string "lettuce leaves"
```

```
    When I put it in upper case
    Then I see the string is "LETTUCE LEAVES"
```

### Steps

```
from lettuce import *
@step('I have the string "(.*)"')
def have_the_string(step, string):
    world.string = string

@step('I put it in upper case')
def put_it_in_upper(step):
    world.string = world.string.upper()

@step('I see the string is "(.*)"')
def see_the_string_is(step, expected):
    assert world.string == expected, \
        "Got %s" % world.string
```

For more information about using Lettuce with Django consult the Lettuce documentation.

### Splinter

From its website: "*Splinter is an open source tool for testing web applications using Python. It lets you automate browser actions, such as visiting URLs and interacting with their items.*"

```
from splinter import Browser
browser = Browser()
# Visit URL
url = "http://www.google.com"
browser.visit(url)
browser.fill('q', 'splinter - python acceptance testing for web applications')
# Find and click the 'search' button
button = browser.find_by_name('btnK')
# Interact with elements
button.click()
if browser.is_text_present('splinter.cobrateam.info'):
    print "Yes, the official website was found!"
else:
    print "No, it wasn't found... We need to improve our SEO techniques"
browser.quit()
```

For more infos about Splinter.

### Selenium

From its website: "*Selenium automates browsers. That's it. What you do with that power is entirely up to you. Primarily it is for automating web applications for testing purposes, but is certainly not limited to just that. Boring web-based administration tasks can (and should!) also be automated as well.*"

For more infos about Selenium.

### Running the tests

See the *Testing and development environment* documentation for an example of how to quickly setup a testing and development environment.

### With Google Chrome

```
cd example_project/

export BROWSER="CHROME"
./run-tests
```

### With Google Firefox

```
export BROWSER="FIREFOX"
./run-tests
```

*Note*: Google Chrome is used as default.

### Test command arguments

If you have special arguments to pass to the test runner you will have to use the full command:

```
python manage.py harvest
```

To test a single feature:

```
python manage.py harvest test_app/features/date.feature
```

Excluding applications:

```
python manage.py harvest -A myApp1,myApp2
```

For a complete argument documentation, please refer to this section of the Lettuce documentation.

### Manual tests

The example_project can also be used to perform manual tests.

While in the virtualenv, use the command *./run-server*. It accepts arguments as usual.

Open the dev server url, an index of the tests should show up.

If you click on a test it will bring you to a page with an URL like this: *http://127.0.0.1:9999/test/char/*.

You can pass arguments to the editlive instance using GET parameters: *http://127.0.0.1:9999/test/char/?class=fixedwidth&width=80&template_filters=upper*.

# Python Module Index

## c



## e

# Index

## Symbols



## A



## B



## C



## D



## E



## F



## G



## I



## J

## M



## R



## S



## T