deform Documentation

Release 0.9.7

Pylons Developers

May 26, 2013

CONTENTS

`deform` is a Python HTML form generation library. It runs under Python 2.6, 2.7, 3.2 and 3.3.

The design of `deform` is heavily influenced by the formish form generation library. Some might even say it's a shameless rip-off; this would not be completely inaccurate. It differs from formish mostly in ways that make the implementation (arguably) simpler and smaller.

`deform` uses *Colander* as a schema library, *Peppercorn* as a form control deserialization library, and *Chameleon* to perform HTML templating.

`deform` depends only on Peppercorn, Colander, Chameleon and an internationalization library named translationstring, so it may be used in most web frameworks (or antiframeworks) as a result.

Alternate templating languages may be used, as long as all templates are translated from the native Chameleon templates to your templating system of choice and a suitable *renderer* is supplied to `deform`.

CHAPTER

ONE

TOPICS

1.1 Basic Usage

In this chapter, we'll walk through basic usage of Deform to render a form, and capture and validate input.

The steps a developer must take to cause a form to be renderered and subsequently be ready to accept form submission input are:

- Define a schema
- Create a form object.
- Assign non-default widgets to fields in the form (optional).
- Render the form.

Once the form is rendered, a user will interact with the form in his browser, and some point, he will submit it. When the user submits the form, the data provided by the user will either validate properly, or the form will need to be rerendered with error markers which help to inform the user of which parts need to be filled in "properly" (as defined by the schema). We allow the user to continue filling in the form, submitting, and revalidating indefinitely.

1.1.1 Defining A Schema

The first step to using Deform is to create a *schema* which represents the data structure you wish to be captured via a form rendering.

For example, let's imagine you want to create a form based roughly on a data structure you'll obtain by reading data from a relational database. An example of such a data structure might look something like this:

```
[
{
 'name':'keith',
 'age':20,
},
{
 'name':'fred',
 'age':23,
},
]
```

In other words, the database query we make returns a sequence of *people*; each person is represented by some data. We need to edit this data. There won't be many people in this list, so we don't need any sort of paging or batching to make our way through the list; we can display it all on one form page.

Deform designates a structure akin to the example above as an *appstruct*. The term "appstruct" is shorthand for "application structure", because it's the kind of high-level structure that an application usually cares about: the data present in an appstruct is useful directly to an application itself.

Note: An appstruct differs from other structures that Deform uses (such as *pstruct* and *cstruct* structures): pstructs and cstructs are typically only useful during intermediate parts of the rendering process.

Usually, given some appstruct, you can divine a *schema* that would allow you to edit the data related to the appstruct. Let's define a schema which will attempt to serialize this particular appstruct to a form. Our application has these requirements of the resulting form:

- It must be possible to add and remove a person.
- It must be possible to change any person's name or age after they've been added.

Here's a schema that will help us meet those requirements:

```
import colander

class Person(colander.MappingSchema):
    name = colander.SchemaNode(colander.String())
    age = colander.SchemaNode(colander.Integer(),
                              validator=colander.Range(0, 200))

class People(colander.SequenceSchema):
    person = Person()

class Schema(colander.MappingSchema):
    people = People()

schema = Schema()
```

The schemas used by Deform come from a package named *Colander*. The canonical documentation for Colander exists at http://docs.pylonsproject.org/projects/colander/dev/ . To compose complex schemas, you'll need to read it to get comfy with the documentation of the default Colander data types. But for now, we can play it by ear.

For ease of reading, we've actually defined *three* schemas above, but we coalesce them all into a single schema instance as `schema` in the last step. A `People` schema is a collection of `Person` schema nodes. As the result of our definitions, a `Person` represents:

- A `name`, which must be a string.
- An `age`, which must be deserializable to an integer; after deserialization happens, a validator ensures that the integer is between 0 and 200 inclusive.

Schema Node Objects

Note: This section repeats and contextualizes the *Colander* documentation about schema nodes in order to prevent you from needing to switch away from this page to another while trying to learn about forms. But you can also get much the same information at http://docs.pylonsproject.org/projects/colander/dev/

A schema is composed of one or more *schema node* objects, each typically of the class `colander.SchemaNode`, usually in a nested arrangement. Each schema node object has a required *type*, an optional *preparer* for adjusting data after deserialization, an optional *validator* for deserialized prepared data, an optional *default*, an optional *missing*, an optional *title*, an optinal *css_class*, an optional *description*, and a slightly less optional *name*. It also accepts *arbitrary* keyword arguments, which are attached directly as attributes to the node instance.

The *type* of a schema node indicates its data type (such as colander.Int or colander.String).

The *preparer* of a schema node is called after deserialization but before validation; it prepares a deserialized value for validation. Examples would be to prepend schemes that may be missing on url values or to filter html provided by a rich text editor. A preparer is not called during serialization, only during deserialization.

The *validator* of a schema node is called after deserialization and preparation ; it makes sure the value matches a constraint. An example of such a validator is provided in the schema above: validator=colander.Range(0, 200). A validator is not called after schema node serialization, only after node deserialization.

The *default* of a schema node indicates the value to be serialized if a value for the schema node is not found in the input data during serialization. It should be the deserialized representation.

The *missing* of a schema node indicates the value to be deserialized if a value for the schema node is not found in the input data during deserialization. It should be the deserialized representation. If a schema node does not have a missing value, a colander.Invalid exception will be raised if the data structure being deserialized does not contain a matching value.

The *name* of a schema node is used to relate schema nodes to each other. It is also used as the title if a title is not provided.

The *title* of a schema node is metadata about a schema node. It shows up in the legend above the form field(s) related to the schema node. By default, it is a capitalization of the *name*.

The *css_class* of a schema node is metadata about a schema node. It shows up as a CSS class on the fieldset, which is rendered from the schema node.

The *description* of a schema node is metadata about a schema node. It shows up as a tooltip when someone hovers over the form control(s) related to a *field*. By default, it is empty.

The name of a schema node that is introduced as a class-level attribute of a colander.MappingSchema, colander.TupleSchema or a colander.SequenceSchema is its class attribute name. For example:

```
import colander

class Phone(colander.MappingSchema):
    location = colander.SchemaNode(colander.String(),
                                   validator=colander.OneOf(['home','work']))
    number = colander.SchemaNode(colander.String())
```

The name of the schema node defined via location = colander.SchemaNode(..) within the schema above is location. The title of the same schema node is Location.

Schema Objects

In the examples above, if you've been paying attention, you'll have noticed that we're defining classes which subclass from colander.MappingSchema, and colander.SequenceSchema. It's turtles all the way down: the result of creating an instance of any of colander.MappingSchema, colander.TupleSchema or colander.SequenceSchema object is *also* a colander.SchemaNode object.

Instantiating a colander.MappingSchema creates a schema node which has a *type* value of colander.Mapping.

Instantiating a colander.TupleSchema creates a schema node which has a *type* value of colander.Tuple.

Instantiating a colander.SequenceSchema creates a schema node which has a *type* value of colander.Sequence.

Creating Schemas Without Using a Class Statement (Imperatively)

See http://docs.pylonsproject.org/projects/colander/dev/basics.html#defining-a-schema-imperatively for information about how to create schemas without using a `class` statement.

Creating a schema with or without `class` statements is purely a style decision; the outcome of creating a schema without `class` statements is the same as creating one with `class` statements.

1.1.2 Rendering a Form

Earlier we defined a schema:

```python
import colander

class Person(colander.MappingSchema):
    name = colander.SchemaNode(colander.String())
    age = colander.SchemaNode(colander.Integer(),
                              validator=colander.Range(0, 200))

class People(colander.SequenceSchema):
    person = Person()

class Schema(colander.MappingSchema):
    people = People()

schema = Schema()
```

Let's now use this schema to create, render and validate a form.

Creating a Form Object

To create a form object, we do this:

```python
from deform import Form
myform = Form(schema, buttons=('submit',))
```

We used the `schema` object (an instance of `colander.MappingSchema`) we created in the previous section as the first positional parameter to the `deform.Form` class; we passed the value `('submit',)` as the value of the `buttons` keyword argument. This will cause a single `submit` input element labeled `Submit` to be injected at the bottom of the form rendering. We chose to pass in the button names as a sequence of strings, but we could have also passed a sequence of instances of the `deform.Button` class. Either is permissible.

Note that the first positional argument to `deform.Form` must be a schema node representing a *mapping* object (a structure which maps a key to a value). We satisfied this constraint above by passing our `schema` object, which we obtained via the `colander.MappingSchema` constructor, as the `schema` argument to the `deform.Form` constructor

Although different kinds of schema nodes can be present in a schema used by a Deform `deform.Form` instance, a form instance cannot deal with a schema node representing a sequence, a tuple schema, a string, an integer, etc. as the value of its `schema` parameter; only a schema node representing a mapping is permissible. This typically means that the object passed as the `schema` argument to a `deform.Form` constructor must be obtained as the result of using the `colander.MappingSchema` constructor (or the equivalent imperative spelling).

Rendering the Form

Once we've created a Form object, we can render it without issue by calling the `deform.Field.render()` method: the `deform.Form` class is a subclass of the `deform.Field` class, so this method is available to a `deform.Form` instance.

If we wanted to render an "add" form (a form without initial data), we'd just omit the appstruct while calling `deform.Field.render()`.

```
form = myform.render()
```

If we have some existing data already that we'd like to edit using the form (the form is an "edit form" as opposed to an "add form"). That data might look like this:

```
appstruct = [
    {
        'name':'keith',
        'age':20,
        },
    {
        'name':'fred',
        'age':23,
        },
    ]
```

To inject it into the serialized form as the data to be edited, we'd pass it in to the `deform.Field.render()` method to get a form rendering:

```
form = myform.render(appstruct)
```

If, finally, instead we wanted to render a "read-only" variant of an edit form using the same appstruct, we'd pass the `readonly` flag as `True` to the `deform.Field.render()` method.

```
form = myform.render(appstruct, readonly=True)
```

This would cause a page to be rendered in a crude form without any form controls, so the user it's presented to cannot edit it.

Once any of the above statements runs, the `form` variable is now a Unicode object containing an HTML rendering of the edit form, useful for serving out to a browser. The root tag of the rendering will be the `<form>` tag representing this form (or at least a `<div>` tag that contains this form tag), so the application using it will need to wrap it in HTML `<html>` and `<body>` tags as necessary. It will need to be inserted as "structure" without any HTML escaping.

Serving up the Rendered Form

We now have an HTML rendering of a form as the variable named `form`. But before we can serve it up successfully to a browser user, we have to make sure that static resources used by Deform can be resolved properly. Some Deform widgets (including at least one we've implied in our sample schema) require access to static resources such as images via HTTP.

For these widgets to work properly, we'll need to arrange that files in the directory named `static` within the `deform` package can be resolved via a URL which lives at the same hostname and port number as the page which serves up the form itself. For example, the URL `/static/css/form.css` should be willing to return the `form.css` CSS file in the `static/css` directory in the `deform` package as `text/css` content and return `/static/scripts/deform.js` as``text/javascript`` content. How you arrange to do this is dependent on your web framework. It's done in `pyramid` imperative configuration via:

```
config = Configurator(...)
...
config.add_static_view('static', 'deform:static')
...
```

Your web framework will use a different mechanism to offer up static files.

Some of the more important files in the set of JavaScript, CSS files, and images present in the `static` directory of the `deform` package are the following:

`static/scripts/jquery-1.4.2.min.js` A local copy of the JQuery javascript library, used by widgets and other JavaScript files.

`static/scripts/deform.js` A JavaScript library which should be loaded by any template which injects a rendered Deform form.

`static/css/form.css` CSS related to form element renderings.

Each of these libraries should be included in the `<head>` tag of a page which renders a Deform form, e.g.:

```
<head>
  <title>
    Deform Demo Site
  </title>
  <!-- Meta Tags -->
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <!-- CSS -->
  <link rel="stylesheet" href="/static/css/form.css" type="text/css" />
  <!-- JavaScript -->
  <script type="text/javascript"
          src="/static/scripts/jquery-1.4.2.min.js"></script>
  <script type="text/javascript"
          src="/static/scripts/deform.js"></script>
</head>
```

The `deform.field.get_widget_resources()` method can be used to tell you which `static` directory-relative files are required by a particular form rendering, so that you can inject only the ones necessary into the page rendering.

The JavaScript function `deform.load()` *must* be called by the HTML page (usually in a script tag near the end of the page, ala `<script..>deform.load()</script>`) which renders a Deform form in order for widgets which use JavaScript to do proper event and behavior binding. If this function is not called, built-in widgets which use JavaScript will not function properly. For example, you might include this within the body of the rendered page near its end:

```
<script type="text/javascript">
   deform.load()
</script>
```

As above, the head should also contain a `<meta>` tag which names a `utf-8` charset in a `Content-Type` http-equiv. This is a sane setting for most systems.

1.1.3 Validating a Form Submission

Once the user seen the form and has chewed on its inputs a bit, he will eventually submit the form. When he submits it, the logic you use to deal with the form validation must do a few things:

- It must detect that a submit button was clicked.
- It must obtain the list of *form controls* from the form POST data.

- It must call the `deform.Form.validate()` method with the list of form controls.
- It must be willing to catch a `deform.ValidationFailure` exception and rerender the form if there were validation errors.

For example, using the *WebOb* API for the above tasks, and the `form` object we created earlier, such a dance might look like this:

```
if 'submit' in request.POST: # detect that the submit button was clicked

    controls = request.POST.items() # get the form controls

    try:
        appstruct = myform.validate(controls)  # call validate
    except ValidationFailure, e: # catch the exception
        return {'form':e.render()} # re-render the form with an exception

    # the form submission succeeded, we have the data
    return {'form':None, 'appstruct':appstruct}
```

The above set of statements is the sort of logic every web app that uses Deform must do. If the validation stage does not fail, a variable named `appstruct` will exist with the data serialized from the form to be used in your application. Otherwise the form will be rerendered.

Note that by default, when any form submit button is clicked, the form will send a post request to the same URL which rendered the form. This can be changed by passing a different `action` to the `deform.Form` constructor.

1.1.4 Seeing it In Action

To see an “add form” in action that follows the schema in this chapter, visit http://deformdemo.repoze.org/sequence_of_mappings/.

To see a “readonly edit form” in action that follows the schema in this chapter, visit http://deformdemo.repoze.org/readonly_sequence_of_mappings/

The application at http://deformdemo.repoze.org is a `pyramid` application which demonstrates most of the features of Deform, including most of the widget and data types available for use within an application that uses Deform.

1.2 Retail Form Rendering

In the previous chapter we demonstrated how to use Deform to render a complete form, including the input fields, the buttons, and so forth. We used the `deform.Field.render()` method, and injected the resulting HTML snippet into a larger HTML page in our application. That is an effective and quick way to put a form on a page, but sometimes you need more fine-grained control over the way form HTML is rendered. For example, you may need form elements to be placed on the page side-by-side or you might need the form’s styling to be radically different than the form styling used by the default rendering of Deform forms. Often it’s easier to use Deform slightly differently, where you do more work yourself to draw the form within a template, and only use Deform for some of its features. We refer to this as “retail form rendering”.

Note: This feature is new as of Deform 0.9.6.

1.2.1 A Basic Example

Our schema and form object:

```
import colander

class Person(colander.MappingSchema):
    name = colander.SchemaNode(colander.String())
    age = colander.SchemaNode(colander.Integer(),
                              validator=colander.Range(0, 200))

schema = Person()
form = deform.Form(schema, resource_registry=resource_registry)
```

We feed the schema into a template as the `form` value. It doesn't matter what kind of templating system you use to do this, but this example will use ZPT. Below, the name `form` refers to the form we just created above:

```
<div class="row"
     tal:repeat="field form">
    <div class="span2">
        ${structure:field.title}
        <span class="req" tal:condition="field.required">*</span>
    </div>
    <div class="span2">
        ${structure:field.serialize()}
    </div>
    <ul tal:condition="field.error">
        <li tal:repeat="error field.error.messages()">
            ${structure:error}
        </li>
    </ul>
</div>
```

The above template iterates over the fields in the form, using the attributes of each field to draw the title.

You can use the `__getitem__` method of a form to grab named form fields instead of iterating over all of its fields. For example:

```
<div tal:define="field form['name']">
    <div class="span2">
        ${structure:field.title}
        <span class="req" tal:condition="field.required">*</span>
    </div>
    <div class="span2">
        ${structure:field.serialize()}
    </div>
    <ul tal:condition="field.error">
        <li tal:repeat="error field.error.messages()">
            ${structure:error}
        </li>
    </ul>
</div>
```

You can use as little or as much of the Deform Field API to draw the widget as you like. The above examples use the `deform.Field.serialize()` method, which is an easy way to let Deform draw the field HTML, but you can draw it yourself instead if you like, and just rely on the field object for its validation errors (if any). Note that the `serialize` method accepts arbitrary keyword arguments that will be passed as top-level arguments to the Deform widget templates, so if you need to change how a particular widget is rendered without doing things completely by hand, you may want to take a look at the existing widget template and see if your need has been anticipated.

In the POST handler for the form, just do things like we did in the last chapter, except if validation fails, just re-render the template with the same form object.

```
controls = request.POST.items() # get the form controls

try:
    appstruct = form.validate(controls)  # call validate
except ValidationFailure, e: # catch the exception
     # .. rerender the form .. its field's .error attributes
     # will be set
```

It is also possible to pass an `appstruct` argument to the `deform.Form` constructor to create "edit forms". Form/field objects are initialized with this appstruct (recursively) when they're created. This means that accessing `form.cstruct` will return the current set of rendering values. This value is reset during validation, so after a validation is done you can re-render the form to show validation errors.

Note that existing Deform widgets are all built using "retail mode" APIs, so if you need examples, you can look at their templates.

Other methods that might be useful during retail form rendering are:

- `deform.Field.__contains__()`
- `deform.Field.start_mapping()`
- `deform.Field.end_mapping()`
- `deform.Field.start_sequence()`
- `deform.Field.end_sequence()`
- `deform.Field.start_rename()`
- `deform.Field.end_rename()`
- `deform.Field.set_appstruct()`
- `deform.Field.set_pstruct()`
- `deform.Field.render_template()`
- `deform.Field.validate_pstruct()` (and the `subcontrol` argument to `deform.Field.validate()`)

1.3 Common Needs

This chapter collects solutions for requirements that will often crop up once you start using Deform for real world applications.

1.3.1 Changing the Default Widget Associated With a Field

Let's take another look at our familiar schema:

```
import colander

class Person(colander.MappingSchema):
    name = colander.SchemaNode(colander.String())
    age = colander.SchemaNode(colander.Integer(),
                              validator=colander.Range(0, 200))

class People(colander.SequenceSchema):
    person = Person()
```

```

class Schema(colander.MappingSchema):
    people = People()

schema = Schema()
```

This schema renders as a *sequence* of *mapping* objects. Each mapping has two leaf nodes in it: a *string* and an *integer*. If you play around with the demo at http://deformdemo.repoze.org/sequence_of_mappings/ you'll notice that, although we don't actually specify a particular kind of widget for each of these fields, a sensible default widget is used. This is true of each of the default types in *Colander*. Here is how they are mapped by default. In the following list, the schema type which is the header uses the widget underneath it by default.

`colander.Mapping`	`deform.widget.MappingWidget`
`colander.Sequence`	`deform.widget.SequenceWidget`
`colander.String`	`deform.widget.TextInputWidget`
`colander.Integer`	`deform.widget.TextInputWidget`
`colander.Float`	`deform.widget.TextInputWidget`
`colander.Decimal`	`deform.widget.TextInputWidget`
`colander.Boolean`	`deform.widget.CheckboxWidget`
`colander.Date`	`deform.widget.DateInputWidget`
`colander.DateTime`	`deform.widget.DateTimeInputWidget`
`colander.Tuple`	`deform.widget.Widget`

Note: Not just any widget can be used with any schema type; the documentation for each widget usually indicates what type it can be used against successfully. If all existing widgets provided by Deform are insufficient, you can use a custom widget. See *Writing Your Own Widget* for more information about writing a custom widget.

If you are creating a schema that contains a type which is not in this list, or if you'd like to use a different widget for a particular field, or you want to change the settings of the default widget associated with the type, you need to associate the field with the widget "by hand". There are a number of ways to do so, as outlined in the sections below.

As an argument to a `colander.SchemaNode` constructor

As of Deform 0.8, you may specify the widget as part of the schema:

```
import colander

from deform import Form
from deform.widget import TextInputWidget

class Person(colander.MappingSchema):
    name = colander.SchemaNode(colander.String(),
                               widget=TextAreaWidget())
    age = colander.SchemaNode(colander.Integer(),
                              validator=colander.Range(0, 200))

class People(colander.SequenceSchema):
    person = Person()

class Schema(colander.MappingSchema):
    people = People()

schema = Schema()

myform = Form(schema, buttons=('submit',))
```

Note above that we passed a `widget` argument to the `name` schema node in the `Person` class above. When a schema containing a node with a `widget` argument to a schema node is rendered by Deform, the widget specified in the node constructor is used as the widget which should be associated with that node's form rendering. In this case, we'll be using a `deform.widget.TextAreaWidget` as the `name` widget.

Note: Widget associations done in a schema are always overridden by explicit widget assigments performed via `deform.Field.__setitem__()` and `deform.Field.set_widgets()`.

Using dictionary access to change the widget

After the `deform.Form` constructor is called with the schema you can change the widget used for a particular field by using dictionary access to get to the field in question. A `deform.Form` is just another kind of `deform.Field`, so the method works for either kind of object. For example:

```
from deform import Form
from deform.widget import TextInputWidget

myform = Form(schema, buttons=('submit',))
myform['people']['person']['name'].widget = TextInputWidget(size=10)
```

This associates the `String` field named `name` in the rendered form with an explicitly created `deform.widget.TextInputWidget` by finding the `name` field via a series of `__getitem__` calls through the field structure, then by assigning an explicit `widget` attribute to the `name` field.

You might want to do this in order to pass a `size` argument to the explicit widget creation, indicating that the size of the `name` input field should be 10em rather than the default.

Although in the example above, we associated the `name` field with the same type of widget as its default we could have just as easily associated the `name` field with a completely different widget using the same pattern. For example:

```
from deform import Form
from deform.widget import TextInputWidget

myform = Form(schema, buttons=('submit',))
myform['people']['person']['name'].widget = TextAreaWidget()
```

The above renders an HTML `textarea` input element for the `name` field instead of an `input type=text` field. This probably doesn't make much sense for a field called `name` (names aren't usually multiline paragraphs); but it does let us demonstrate how different widgets can be used for the same field.

Using the `deform.Field.set_widgets()` method

Equivalently, you can also use the `deform.Field.set_widgets()` method to associate multiple widgets with multiple fields in a form. For example:

```
from deform import Form
from deform.widget import TextInputWidget

myform = Form(schema, buttons=('submit',))
```

```
myform.set_widgets({'people.person.name':TextAreaWidget(),
                    'people.person.age':TextAreaWidget()})
```

Each key in the dictionary passed to `deform.Field.set_widgets()` is a "dotted name" which resolves to a single field element. Each value in the dictionary is a widget instance. See `deform.Field.set_widgets()` for more information about this method and dotted name resolution, including special cases which involve the "splat" (`*`) character and the empty string as a key name.

1.3.2 Using Text Input Masks

The `deform.widget.TextInputWidget` and `deform.widget.CheckedInputWidget` widgets allow for the use of a fixed-length text input mask. Use of a text input mask causes placeholder text to be placed in the text field input, and restricts the type and length of the characters input into the text field.

For example:

When using a text input mask:

`a` represents an alpha character (A-Z,a-z)

`9` represents a numeric character (0-9)

`*` represents an alphanumeric character (A-Z,a-z,0-9)

All other characters in the mask will be considered mask literals.

By default the placeholder text for non-literal characters in the field will be _ (the underscore character). To change this for a given input field, use the `mask_placeholder` argument to the TextInputWidget:

```
form['date'].widget = TextInputWidget(mask='99/99/9999',
                                      mask_placeholder="-")
```

Example masks:

Date 99/99/9999

US Phone

999. 999-9999

US SSN 999-99-9999

When this option is used, the *jquery.maskedinput* library must be loaded into the page serving the form for the mask argument to have any effect. A copy of this library is available in the `static/scripts` directory of the `deform` package itself.

See http://deformdemo.repoze.org/text_input_masks/ for a working example.

Use of a text input mask is not a replacement for server-side validation of the field; it is purely a UI affordance. If the data must be checked at input time a separate *validator* should be attached to the related schema node.

1.3.3 Using the AutocompleteInputWidget

The `deform.widget.AutocompleteInputWidget` widget allows for client side autocompletion from provided choices in a text input field. To use this you **MUST** ensure that *jQuery* and the *JQuery UI* plugin are available to the page where the `deform.widget.AutocompleteInputWidget` widget is rendered.

For convenience a version of the *JQuery UI* (which includes the `autocomplete` sublibrary) is included in the `deform` static directory. Additionally, the *JQuery UI* styles for the selection box are also included in the `deform` `static` directory. See *Serving up the Rendered Form* and *The (High-Level) deform.Field.get_widget_resources() Method* for more information about using the included libraries from your application.

A very simple example of using `deform.widget.AutocompleteInputWidget` follows:

```
form[’frozznobs’].widget = AutocompleteInputWidget(
                             values=[’spam’, ’eggs’, ’bar’, ’baz’])
```

Instead of a list of values a URL can be provided to values:

```
form[’frobsnozz’].widget = AutocompleteInputWidget(
                             values=’http://example.com/someapi’)
```

In the above case a call to the url should provide results in a JSON-compatible format or JSONP-compatible response if on a different host than the application. Something like either of these structures in JSON are suitable:

```
//Items are used as both value and label
[’item-one’, ’item-two’, ’item-three’]

//Separate values and labels
[
    {’value’: ’item-one’, ’label’: ’Item One’},
    {’value’: ’item-two’, ’label’: ’Item Two’},
    {’value’: ’item-three’, ’label’: ’Item Three’}
]
```

The autocomplete plugin will add a query string to the request URL with the variable `term` which contains the user’s input at that momement. The server may use this to filter the returned results.

For more information, see http://api.jqueryui.com/autocomplete/#option-source - specifically, the section concerning the `String` type for the `source` option.

Some options for the *jquery.autocomplete* plugin are mapped and can be passed to the widget. See `deform.widget.AutocompleteInputWidget` for details regarding the available options. Passing options looks like:

```
form[’nobsfrozz’].widget = AutocompleteInputWidget(
                             values=[’spam, ’eggs’, ’bar’, ’baz’],
                             min_length=1)
```

See http://deformdemo.repoze.org/autocomplete_input/ and http://deformdemo.repoze.org/autocomplete_remote_input/ for working examples. A working example of a remote URL providing completion data can be found at http://deformdemo.repoze.org/autocomplete_input_values/.

Use of `deform.widget.AutocompleteInputWidget` is not a replacement for server-side validation of the field; it is purely a UI affordance. If the data must be checked at input time a separate *validator* should be attached to the related schema node.

1.3.4 Creating a New Schema Type

Sometimes the default schema types offered by Colander may not be sufficient to model all the structures in your application.

If this happens, refer to the Colander documentation on *Defining a New Type*.

1.4 Deform Components

A developer dealing with Deform has to understand a few fundamental types of objects and their relationships to one another. These types are:

- schema nodes
- field objects
- widgets

1.4.1 The Relationship Between Widgets, Fields, and Schema Objects

The relationship between widgets, fields, and schema node objects is as follows:

- A schema is created by a developer. It is a collection of *schema node* objects.
- When a root schema node is passed to the `deform.Form` constructor, the result is a *field* object. For each node defined by the developer in the schema recursively, a corresponding *field* is created.
- Each field in the resulting field tree has a default widget type. If the `widget` attribute of a field object is not set directly by the developer, a property is used to create an instance of the default widget type when `field.widget` is first requested. Sane defaults for each schema type typically exist; if a sane default cannot be found, the `deform.widget.TextInputWidget` widget is used.

Note: The Colander documentation is a resource useful to Deform developers. In particular, it details how a *schema* is created and used. Deform schemas are Colander schemas. The Colander documentation about how they work applies to creating Deform schemas as well.

A widget is related to one or more *schema node* type objects. For example, a notional "TextInputWidget" may be responsible for serializing textual data related to a schema node which has `colander.String` as its type into a text input control, while a notional "MappingWidget" might be responsible for serializing a `colander.Mapping` object into a sequence of controls. In both cases, the data type being serialized is related to the schema node type to which the widget is related.

A widget has a relationship to a schema node via a *field* object. A *field* object has a reference to both a widget and a *schema node*. These relationships look like this:

```
field object (``field``)
       |
       |
       |----- widget object  (``field.widget``)
       |
       |
       \----- schema node object (``field.schema``)
```

1.5 Serialization and Deserialization

Serialization is the act of converting application data into a form rendering. Deserialization is the act of converting data resulting from a form submission into application data.

1.5.1 Serialization

Serialization is what happens when you ask Deform to render a form given a *schema*. Here's a high-level overview of what happens when you ask Deform to do this:

- For each *schema node* in the *schema* provided by the application developer, Deform creates a *field*. This happens recursively for each node in the schema. As a result, a tree of fields is created, mirroring the nodes in the schema.
- Each field object created as a result of the prior step knows about its associated schema node (it has a `field.schema` attribute); each field also knows about an associated *widget* object (it has a `field.widget` attribute). This widget object may be a default widget based on the schema node type or it might be overridden by the application developer for a particular rendering.
- Deform passes an *appstruct* to the root schema node's `serialize` method to obtain a *cstruct*. The root schema node is responsible for consulting its children nodes during this process to serialilize the entirety of the data into a single *cstruct*.
- Deform passes the resulting *cstruct* to the root widget object's `serialize` method to generate an HTML form rendering. The root widget object is responsible for consulting its children nodes during this process to serialilize the entirety of the data into an HTML form.

If you were to attempt to produce a high-level overview diagram this process, it might look like this:

```
appstruct -> cstruct -> form
           |          |
           v          v
        schema      widget
```

Peppercorn Structure Markers

You'll see the default deform widget "serializations" (form renderings) make use of *Peppercorn structure markers*.

Peppercorn is a library that is used by Deform; it allows Deform to treat the *form controls* in an HTML form submission as a *stream* instead of a flat mapping of name to value. To do so, it uses hidden form elements to denote structure.

Peppercorn structure markers come in pairs which have a begin token and an end token. For example, a given form rendering might have a part that looks like so:

```
<html>
 ...
  <input type="hidden" name="__start__" value="date:mapping"/>
  <input name="day"/>
  <input name="month"/>
  <input name="year"/>
  <input type="hidden" name="__end__"/>
 ...
</html>
```

The above example shows an example of a pair of peppercorn structure markers which begin and end a *mapping*. The example uses this pair to means that a the widget related to the *date* node in the schema will be be passed a *pstruct* that is a dictionary with multiple values during deserialization: the dictionary will include the keys `day` , `month`, and `year`, and the values will be the values provided by the person interacting with the related form controls.

Other uses of Peppercorn structure markers include: a "confirm password" widget can render a peppercorn mapping with two text inputs in it, a "mapping widget" can serve as a substructure for a fieldset. Basically, Peppercorn makes it more pleasant to deal with form submission data by pre-converting the data from a flat mapping into a set of mappings, sequences, and strings during deserialization.

However, if a widget doesn't want to do anything fancy and a particular widget is completely equivalent to one form control, it doesn't need to use any Peppercorn structure markers in its rendering.

Note: See the Peppercorn documentation for more information about using peppercorn structure markers in HTML.

1.5.2 Deserialization

High-level overview of how "deserialization" (converting form control data resulting from a form submission to application data) works:

- For each *schema node* in the *schema* provided by the application developer, Deform creates a *field*. This happens recursively for each node in the schema. As a result, a tree of fields is created, mirroring the nodes in the schema.
- Each field object created as a result of the prior step knows about its associated schema node (it has a `field.schema` attribute); each field also knows about an associated *widget* object (it has a `field.widget` attribute). This widget object may be a default widget based on the schema node type or it might be overridden by the application developer for a particular rendering.
- Deform passes a set of *form controls* to the `parse` method of *Peppercorn* in order to obtain a *pstruct*.
- Deform passes the resulting *pstruct* to the root widget node's `deserialize` method in order to generate a *cstruct*.
- Deform passes the resulting *cstruct* to the root schema node's `deserialize` method to generate an *appstruct*. This may result in a validation error. If a validation error occurs, the form may be rerendered with error markers in place.

If you were to attempt to produce a high-level overview diagram this process, it might look like this:

```
formcontrols -> pstruct -> cstruct -> appstruct
             |          |          |
             v          v          v
         peppercorn   widget     schema
```

When a user presses the submit button on any Deform form, Deform itself runs the resulting *form controls* through the `peppercorn.parse` method. This converts the form data into a mapping. The *structure markers* in the form data indicate the internal structure of the mapping.

For example, if the form submitted had the following data:

```
<html>
 ...
  <input type="hidden" name="__start__" value="date:mapping"/>
  <input name="day"/>
  <input name="month"/>
  <input name="year"/>
  <input type="hidden" name="__end__"/>
 ...
</html>
```

There would be a `date` key in the root of the pstruct mapping which held three keys: `day`, `month`, and `year`.

Note: See the Peppercorn documentation for more information about the result of the `peppercorn.parse` method and how it relates to form control data.

The bits of code that are "closest" to the browser are called "widgets". A chapter about creating widgets exists in this documentation at *Writing Your Own Widget*.

A widget has a `deserialize` method. The deserialize method is passed a structure (a *pstruct*) which is shorthand for “Peppercorn structure”. A *pstruct* might be a string, it might be a mapping, or it might be a sequence, depending on the output of `peppercorn.parse` related to its schema node against the form control data.

The job of the deserialize method of a widget is to convert the pstruct it receives into a *cstruct*. A *cstruct* is a shorthand for “Colander structure”. It is often a string, a mapping or a sequence.

An application eventually wants to deal in types less primitive than strings: a model instance or a datetime object. An *appstruct* is the data that an application that uses Deform eventually wants to deal in. Therefore, once a widget has turned a *pstruct* into a *cstruct*, the *schema node* related to that widget is responsible for converting that cstruct to an *appstruct*. A schema node possesses its very own `deserialize` method, which is responsible for accepting a *cstruct* and returning an *appstruct*.

Raising Errors During Deserialization

If a widget determines that a pstruct value cannot be converted successfully to a cstruct value during deserialization, it may raise an `colander.Invalid` exception.

When it raises this exception, it can use the field object as a “scratchpad” to hold on to other data, but it must pass a `value` attribute to the exception constructor. For example:

```
import colander

def serialize(self, field, cstruct, readonly=False):
    if cstruct is colander.null:
        cstruct = ''
    confirm = getattr(field, 'confirm', '')
    template = readonly and self.readonly_template or self.template
    return field.renderer(template, field=field, cstruct=cstruct,
                          confirm=confirm, subject=self.subject,
                          confirm_subject=self.confirm_subject,
                          )

def deserialize(self, field, pstruct):
    if pstruct is colander.null:
        return colander.null
    value = pstruct.get('value') or ''
    confirm = pstruct.get('confirm') or ''
    field.confirm = confirm
    if value != confirm:
        raise Invalid(field.schema, self.mismatch_message, value)
    return value
```

The schema type associated with this widget is expecting a single string as its cstruct. The `value` passed to the exception constructor raised during the `deserialize` when `value != confirm` is used as that `cstruct` value when the form is rerendered with error markers. The `confirm` value is picked off the field value when the form is rerendered at this time.

1.5.3 Say What?

Q: “So deform colander and peppercorn are pretty intertwingled?”

A: “Colander and Peppercorn are unrelated; Deform is effectively something that integrates colander and peppercorn together.”

1.6 Templates

A set of *Chameleon* templates is used by the default widget set present in deform to make it easier to customize the look and feel of form renderings.

1.6.1 Overriding the default templates

The default widget set uses templates that live in the templates directory of the deform package. If you are comfortable using the *Chameleon* templating system, but you simply need to override some of these templates you can create your own template directory and copy the template you wish to customize into it. You can then either configure your new template directory to be used for all forms or for specific forms as described below.

For relevant API documentation see the deform.ZPTRendererFactory class and the deform.Field class renderer argument.

Overriding for all forms

To globally override templates use the deform.Field.set_zpt_renderer() class method to change the settings associated with the default ZPT renderer:

```
from pkg_resources import resource_filename
from deform import Form

deform_templates = resource_filename('deform', 'templates')
search_path = ('/path/to/my/templates', deform_templates)

Form.set_zpt_renderer(search_path)
```

Now, the templates in /path/to/my/templates will be used in preference to the default templates whenever a form is rendered. Any number of template directories can be put into the search path and will be searched in the order specified with the first matching template found being used.

Overriding for specific forms

If you only want to change the templates used for a specific form, or even for the specific rendering of a form, you can pass a renderer argument to the deform.Form constructor, e.g.:

```
from deform import ZPTRendererFactory
from deform import Form
from pkg_resources import resource_filename

deform_templates = resource_filename('deform', 'templates')
search_path = ('/path/to/my/templates', deform_templates)
renderer = ZPTRendererFactory(search_path)

form = Form(someschema, renderer=renderer)
```

When the above form is rendered, the templates in /path/to/my/templates will be used in preference to the default templates. Any number of template directories can be put into the search path and will be searched in the order specified with the first matching template found being used.

1.6.2 Using an alternative templating system

A *renderer* is used by the each widget implementation in `deform` to render HTML from a set of templates. By default, each of the default Deform widgets uses a template written in the Chameleon ZPT templating language. If you'd rather use a different templating system for your widgets, you can. To do so, you need to:

- Write an alternate renderer that uses the templating system of your choice.
- Optionally, convert all the existing Deform templates to your templating language of choice. This is only necessary if you choose to use the widgets that ship as part of Deform.
- Set the default renderer of the `deform.Form` class.

Creating a Renderer

A renderer is simply a callable that accepts a single positional argument, which is the template name, and a set of keyword arguments. The keyword arguments it will receive are arbitrary, and differ per widget, but the keywords usually include `field`, a *field* object, and `cstruct`, the data structure related to the field that must be rendered by the template itself.

Here's an example of a (naive) renderer that uses the Mako templating engine:

```
from mako.template import Template

def mako_renderer(tmpl_name, **kw):
    template = Template(filename='/template_dir/%s.mak' % tmpl_name)
    return template.render(**kw)
```

Note: A more robust implementation might use a template loader that does some caching, or it might allow the template directory to be configured.

Note the `mako_renderer` function we've created actually appends a `.mak` extension to the `tmpl_name` it is passed. This is because Deform passes a template name without any extension to allow for different templating systems to be used as renderers.

Our `mako_renderer` renderer is now ready to have some templates created for it.

Converting the Default Deform Templates

The `deform` package contains a directory named `templates`. You can see the current trunk contents of this directory by browsing the source repository. Each file within this directory and any of its subdirectories is a Chameleon ZPT template that is used by a default Deform widget.

For example, `textinput.pt` ZPT template, which is used by the `deform.widget.TextInputWidget` widget and which renders a text input control looks like this:

```
<span tal:define="name name|field.name;
                  size size|field.widget.size;
                  css_class css_class|field.widget.css_class;
                  oid oid|field.oid;
                  mask mask|field.widget.mask;
                  mask_placeholder mask_placeholder|field.widget.mask_placeholder;
                  style style|field.widget.style|None;
"
      tal:omit-tag="">
    <input type="text" name="${name}" value="${cstruct}"
```

```
           tal:attributes="size size;
                           class css_class;
                           style style"
           id="${oid}"/>
    <script tal:condition="mask" type="text/javascript">
      deform.addCallback(
         '${oid}',
         function (oid) {
            $("#" + oid).mask("${mask}",
                 {placeholder:"${mask_placeholder}"});
         });
    </script>
</span>
```

If we created a Mako renderer, we would need to create an analogue of this template. Such an analogue should be named `textinput.mak` and might look like this:

```
<input type="text" name="${field.name}" value="${cstruct}"
% if field.widget.size:
size=${field.widget.size}
% endif
/>
```

Whatever the body of the template looks like, the resulting `textinput.mak` should be placed in a directory that is meant to house other Mako template files which are going to be consumed by Deform. You'll need to convert each of the templates that exist in the Deform `templates` directory and its subdirectories, and put all of the resulting templates into your private mako `templates` dir too, retaining any directory structure (e.g., retaining the fact that there is a `readonly` directory and converting its contents).

Configuring Your New Renderer as the Default

Once you've created a new renderer and created templates that match all the existing Deform templates, you can now configure your renderer to be used by Deform. In startup code, add something like:

```
from mymakorenderer import mako_renderer

from deform import Form
Form.set_default_renderer(mako_renderer)
```

The deform widget system will now use your renderer as the default renderer.

Note that calling `deform.Field.set_default_renderer()` will cause this renderer to be used by default by all consumers in the process it's invoked in. This is potentially undesirable: you may need the same process to use more than one renderer perhaps because that same process houses two different Deform-using systems. In this case, instead of using the `set_default_renderer` method, you can write your application in such a way that it passes a renderer to the Form constructor:

```
from mymakorenderer import mako_renderer
from deform import Form

...
schema = SomeSchema()
form = Form(schema, renderer=mako_renderer)
```

1.7 Widgets

A widget is a bit of code that is willing to:

- serialize a *cstruct* into HTML for display in a form rendering
- deserialize data obtained from a form post (a *pstruct*) into a data structure suitable for deserialization by a *schema node* (a *cstruct*).
- handle validation errors

Deform ships with a number of built-in widgets. You hopefully needn't create your own widget unless you're trying to do something that the built-in widget set didn't anticipate. However, when a built-in Deform widget doesn't do exactly what you want, you can extend Deform by creating a new widget that is more suitable for the task.

1.7.1 Widget Templates

A widget needn't use a template file, but each of the built-in widgets does. A template is usually assigned to a default widget via its `template` and `readonly_template` attributes; those attributes are then used in the `serialize` method of the widget, ala:

```
def serialize(self, field, cstruct, readonly=False):
    if cstruct in (null, None):
        cstruct = ’’
    template = readonly and self.readonly_template or self.template
    return field.renderer(template, field=field, cstruct=cstruct)
```

The `deform.field.renderer()` method is a method which accepts a logical template name (such as `texinput`) and renders it using the active Deform *renderer*; the default renderer is the ZPT renderer, which uses the templates within the `deform/templates` directory within the `deform` package. See *Templates* for more information about widget templates.

1.7.2 Widget Javascript

Some built-in Deform widgets require JavaScript. In order for the built-in Deform widgets that require JavaScript to function properly, the `deform.load()` JavaScript function must be called when the page containing a form is renderered.

Some built-in Deform widgets include JavaScript which operates against a local input element when it is loaded. For example, the `deform.widget.AutocompleteInputWidget` template looks like this:

```
<span tal:omit-tag="">
    <input type="text"
           name="${field.name}"
           value="${cstruct}"
           tal:attributes="size field.widget.size;
                           class field.widget.css_class"
           id="${field.oid}"/>
    <script tal:condition="field.widget.values" type="text/javascript">
      deform.addCallback(
        ’${field.oid}’,
        function (oid) {
            $(’#’ + oid).autocomplete({source: ${values}});
            $(’#’ + oid).autocomplete("option", ${options});
        }
      );
    </script>
</span>
```

field.oid refers to the ordered identifier that Deform gives to each field widget rendering. You can see that the script which runs when this widget is included in a rendering calls a function named deform.addCallback, passing it the value of field.oid and a callback function as oid and callback respectively. When it is executed, the callback function calls the autocomplete method of the JQuery selector result for $(’#’ + oid).

The callback define above will be called under two circumstances:

- When the page first loads and the deform.load() JavaScript function is called.
- When a *sequence* is involved, and a sequence item is added, resulting in a call to the deform.addSequenceItem() JavaScript function.

The reason that default Deform widgets call deform.addCallback rather than simply using ${field.oid} directly in the rendered script is becase sequence item handling happens entirely client side by cloning an existing prototype node, and before a sequence item can be added, all of the id attributes in the HTML that makes up the field must be changed to be unique. The addCallback indirection assures that the callback is executed with the *modified* oid rather than the protoype node’s oid. Your widgets should do the same if they are meant to be used as part of sequences.

1.7.3 Widget Requirements and Resources

Some widgets require external resources to work properly (such as CSS and Javascript files). Deform provides mechanisms that will allow you to determine *which* resources are required by a particular form rendering, so that your application may include them in the HEAD of the page which includes the rendered form.

The (Low-Level) deform.Field.get_widget_requirements() Method

After a form has been fully populated with widgets, the deform.Field.get_widget_requirements() method called on the form object will return a sequence of two-tuples. When a non-empty sequence is returned by deform.Field.get_widget_requirements(), it means that one or more CSS or JavaScript resources will need to be loaded by the page performing the form rendering in order for some widget on the page to function properly.

The first element in each two-tuple represents a *requirement name*. It represents a logical reference to one *or more* Javascript or CSS resources. The second element in each two-tuple is the reqested version of the requirement. It may be None, in which case the version required is unspecified. When the version required is unspecified, a default version of the resource set will be chosen.

The requirement name / version pair implies a set of resources, but it is not a URL, nor is it a filename or a filename prefix. The caller of deform.Field.get_widget_requirements() must use the resource names returned as *logical* references. For example, if the requirement name is jquery, and the version id is 1.4.2, the caller can take that to mean that the JQuery library should be loaded within the page header via, for example the inclusion of the HTML <script type="text/javascript" src="http://deformdemo.repoze.org/static/scripts/jquery-1.4.2.min.js"></script> within the HEAD tag of the rendered HTML page.

Users will almost certainly prefer to use the deform.Field.get_widget_resources() API (explained in the succeeding section) to obtain a fully expanded list of relative resource paths required by a form rendering. deform.Field.get_widget_requirements(), however, may be used if custom requirement name to resource mappings need to be done without the help of a *resource registry*.

See also the description of requirements in deform.Widget.

The (High-Level) `deform.Field.get_widget_resources()` Method

A mechanism to resolve the requirements of a form into relative resource filenames exists as a method: `deform.Field.get_widget_resources()`.

Note: Because Deform is framework-agnostic, this method only *reports* to its caller the resource paths required for a successful form rendering, it does not (cannot) arrange for the reported requirements to be satisfied in a page rendering; satisfying these requirements is the responsibility of the calling code.

The `deform.Field.get_widget_resources()` method returns a dictionary with two keys: `js` and `css`. The value related to each key in the dictionary is a list of *relative* resource names. Each resource name is assumed to be relative to the static directory which houses your application's Deform resources (usually a copy of the `static` directory inside the Deform package). If the method is called with no arguments, it will return a dictionary in the same form representing resources it believes are required by the current form. If it is called with a set of requirements (the value returned by the `deform.Field.get_widget_requirements()` method), it will attempt to resolve the requirements passed to it. You might use it like so:

```
import deform

form = deform.Form(someschema)
resources = form.get_widget_resources()
js_resources = resources['js']
css_resources = resources['css']
js_links = [ 'http://my.static.place/%s' % r for r in js_resources ]
css_links = [ 'http://my.static.place/%s' % r for r in css_resources ]
js_tags = ['<script type="text/javascript" src="%s"></script>' % link
           for link in js_links]
css_tags = ['<link rel="stylesheet" href="%s"/>' % link
           for link in css_links]
tags = js_tags + css_tags
return {'form':form.render(), 'tags':tags}
```

The template rendering the return value would need to make sense of "tags" (it would inject them wholesale into the HEAD). Obviously, other strategies for rendering HEAD tags can be devised using the result of `get_widget_resources`, this is just an example.

`deform.Field.get_widget_resources()` uses a *resource registry* to map requirement names to resource paths. If `deform.Field.get_widget_resources()` cannot resolve a requirement name, or it cannot find a set of resources related to the supplied *version* of the requirement name, an `ValueError` will be raised. When this happens, it means that the *resource registry* associated with the form cannot resolve a requirement name or version. When this happens, a resource registry that knows about the requirement will need to be associated with the form explicitly, e.g.:

```
registry = deform.widget.ResourceRegistry()
registry.set_js_resources('requirement', 'ver', 'bar.js', 'baz.js')
registry.set_css_resources('requirement', 'ver', 'foo.css', 'baz.css')

form = Form(schema, resource_registry=registry)
resources = form.get_widget_resources()
js_resources = resources['js']
css_resources = resources['css']
js_links = [ 'http://my.static.place/%s' % r for r in js_resources ]
css_links = [ 'http://my.static.place/%s' % r for r in css_resources ]
js_tags = ['<script type="text/javascript" src="%s"></script>' % link
           for link in js_links]
css_tags = ['<link type="text/css" href="%s"/>' % link
           for link in css_links]
```

```
tags = js_tags + css_tags
return {'form':form.render(), 'tags':tags}
```

An alternate default resource registry can be associated with *all* forms by calling the `deform.Field.set_default_resource_registry()` class method:

```
registry = deform.widget.ResourceRegistry()
registry.set_js_resources('requirement', 'ver', 'bar.js', 'baz.js')
registry.set_css_resources('requirement', 'ver', 'foo.css', 'baz.css')
Form.set_default_resource_registry(registry)
```

This will result in the `registry` registry being used as the default resource registry for all form instances created after the call to `set_default_resource_registry`, hopefully allowing resource resolution to work properly again.

See also the documentation of the `resource_registry` argument in `deform.Field` and the documentation of `deform.widget.ResourceRegistry`.

Specifying Widget Requirements

When creating a new widget, you may specify its requirements by using the `requirements` attribute:

```
from deform.widget import Widget

class MyWidget(Widget):
    requirements = ( ('jquery', '1.4.2'), )
```

There are no hard-and-fast rules about the composition of a requirement name. Your widget's docstring should explain what its requirement names mean, and how map to the logical requirement name to resource paths within a a *resource registry*. For example, your docstring might have text like this: "This widget uses a library name of `jquery.tools` in its requirements list. The name `jquery.tools` implies that the JQuery Tools library must be loaded before rendering the HTML page containing any form which uses this widget; JQuery Tools depends on JQuery, so JQuery should also be loaded. The widget expects JQuery Tools version X.X (as specified in the version field), which expects JQuery version X.X to be loaded previously.". It might go on to explain that a set of resources need to be added to a *resource registry* in order to resolve the logical `jquery.tools` name to a set of relative resource paths, and that the resulting custom resource registry should be used when constructing the form. The default resource registry (`deform.widget.resource_registry`) does not contain resource mappings for your newly-created requirement.

1.7.4 Writing Your Own Widget

Writing a Deform widget means creating an object that supplies the notional Widget interface, which is described in the `deform.widget.Widget` class documentation. The easiest way to create something that implements this interface is to create a class which inherits directly from the `deform.widget.Widget` class itself.

The `deform.widget.Widget` class has a concrete implementation of a constructor and the `handle_error` method as well as default values for all required attributes. The `deform.widget.Widget` class also has abstract implementations of `serialize` and `deserialize` each of which which raises a `NotImplementedError` exception; these must be overridden by your subclass; you may also optionally override the `handle_error` method of the base class.

For example:

```
 from deform.widget import Widget

 class MyInputWidget(Widget):
    def serialize(self, field, cstruct=None, readonly=False):
        ...

    def deserialize(self, field, pstruct=None):
        ...

    def handle_error(self, field, error):
        ...
```

We describe the `serialize`, `deserialize` and `handle_error` methods below.

The `serialize` Method

The `serialize` method of a widget must serialize a *cstruct* value to an HTML rendering. A *cstruct* value is the value which results from a *Colander* schema serialization for the schema node associated with this widget. The result of this method should always be a `unicode` type containing some HTML.

The `field` argument passed to `serialize` is the *field* object to which this widget is attached. Because a *field* object itself has a reference to the widget it uses (as `field.widget`), the field object is passed to the `serialize` method of the widget rather than the widget having a `field` attribute in order to avoid a circular reference.

If the `readonly` argument passed to `serialize` is `True`, it indicates that the result of this serialization should be a read-only rendering (no active form controls) of the `cstruct` data to HTML.

Let's pretend our new `MyInputWidget` only needs to create a text input control during serialization. Its `serialize` method might get defined as so:

```
from deform.widget import Widget
from colander import null
import cgi

class MyInputWidget(Widget):
    def serialize(self, field, cstruct=None, readonly=False):
        if cstruct is null:
            cstruct = u''
        quoted = cgi.escape(cstruct, quote='"')
        return u'<input type="text" value="%s">' % quoted
```

Note that every `serialize` method is responsible for returning a serialization, no matter whether it is provided data by its caller or not. Usually, the value of `cstruct` will contain appropriate data that can be used directly by the widget's rendering logic. But sometimes it will be `colander.null`. It will be `colander.null` when a form which uses this widget is serialized without any data; for example an "add form".

All widgets *must* check if the value passed as `cstruct` is the `colander.null` sentinel value during `serialize`. Widgets are responsible for handling this eventuality, often by serializing a logically "empty" value.

Regardless of how the widget attempts to compute the default value, it must still be able to return a rendering when `cstruct` is `colander.null`. In the example case above, the widget uses the empty string as the `cstruct` value, which is appropriate for this type of "scalar" input widget; for a more "structural" kind of widget the default might be something else like an empty dictionary or list.

The `MyInputWidget` we created in the example does not use a template. Any widget may use a template, but using one is not required; whether a particular widget uses a template is really none of Deform's business: deform simply expects a widget to return a Unicode object containing HTML from the widget's `serialize` method; it doesn't really much care how the widget creates that Unicode object.

Each of the built-in Deform widgets (the widget implementations in `deform.widget`) happens to use a template in order to make it easier for people to override how each widget looks when rendered without needing to change Deform-internal Python code. Instead of needing to change the Python code related to the widget itself, users of the built-in widgets can often perform enough customization by replacing the template associated with the built-in widget implementation. However, this is purely a convenience; templates are largely a built-in widget set implementation detail, not an integral part of the core Deform framework.

Note that “scalar” widgets (widgets which represent a single value as opposed to a collection of values) are not responsible for providing “page furniture” such as a “Required” label or a surrounding div which is used to provide error information when validation fails. This is the responsibility of the “structural” widget which is associated with the parent field of the scalar widget’s field (the “parent widget”); the parent widget is usually one of `deform.widget.MappingWidget` or `deform.widget.SequenceWidget`.

The `deserialize` Method

The `deserialize` method of a widget must deserialize a *pstruct* value to a *cstruct* value and return the *cstruct* value. The `pstruct` argument is a value resulting from the `parse` method of the *Peppercorn* package. The `field` argument is the field object to which this widget is attached.

```python
from deform.widget import Widget
from colander import null
import cgi

class MyInputWidget(Widget):
    def serialize(self, field, cstruct, readonly=False):
        if cstruct is null:
            cstruct = u''
        return '<input type="text" value="%s">' % cgi.escape(cstruct)

    def deserialize(self, field, pstruct):
        if pstruct is null:
            return null
        return pstruct
```

Note that the `deserialize` method of a widget must, like `serialize`, deal with the possibility of being handed a `colander.null` value. `colander.null` will be passed to the widget when a value is missing from the pstruct. The widget usually handles being passed a `colander.null` value in `deserialize` by returning *colander.null*‘, which signifies to the underlying schema that the default value for the schema node should be used if it exists.

The only other real constraint of the deserialize method is that the `serialize` method must be able to reserialize the return value of `deserialize`.

The `handle_error` Method

The `deform.widget.Widget` class already has a suitable implementation; if you subclass from `deform.widget.Widget`, overriding the default implementation is not necessary unless you need special error-handling behavior.

Here’s an implementation of the `deform.widget.Widget.handle_error()` method in the MyInputWidget class:

```python
from deform.widget import Widget
from colander import null
import cgi

class MyInputWidget(Widget):
    def serialize(self, field, cstruct, readonly=False):
        if cstruct is null:
            cstruct = u''
        return '<input type="text" value="%s">' % cgi.escape(cstruct)

    def deserialize(self, field, pstruct):
        if pstruct is null:
            return null
        return pstruct

    def handle_error(self, field, error):
        if field.error is None:
            field.error = error
        for e in error.children:
            for num, subfield in enumerate(field.children):
                if e.pos == num:
                    subfield.widget.handle_error(subfield, e)
```

The `handle_error` method of a widget must:

- Set the `error` attribute of the `field` object it is passed if the `error` attribute has not already been set.
- Call the `handle_error` methods of any subfields which also have errors.

The ability to override `handle_error` exists purely for advanced tasks, such as presenting all child errors of a field on a parent field. For example:

```python
def handle_error(self, field, error):
    msgs = []
    if error.msg:
        field.error = error
    else:
        for e in error.children:
            msgs.append('line %s: %s' % (e.pos+1, e))
        field.error = Invalid(field.schema, '\n'.join(msgs))
```

This implementation does not attach any errors to field children; instead it attaches all of the child errors to the field itself for review.

The Template

The template you use to render a widget will receive input from the widget object, including `field`, which will be the field object represented by the widget. It will usually use the `field.name` value as the `name` input element of the primary control in the widget, and the `field.oid` value as the `id` element of the primary control in the widget.

1.8 Example App

An example is worth a thousand words. Here's an example Pyramid application demonstrating how one might use `deform` to render a form.

> **Warning:** `deform` is not dependent on `pyramid` at all; we use Pyramid in the examples below only to facilitate demonstration of an actual end-to-end working application that uses Deform.

Here's the Python code:

```python
import os

from paste.httpserver import serve
from pyramid.config import Configurator

from colander import MappingSchema
from colander import SequenceSchema
from colander import SchemaNode
from colander import String
from colander import Boolean
from colander import Integer
from colander import Length
from colander import OneOf

from deform import ValidationFailure
from deform import Form
from deform import widget


here = os.path.dirname(os.path.abspath(__file__))

colors = ((’red’, ’Red’), (’green’, ’Green’), (’blue’, ’Blue’))

class DateSchema(MappingSchema):
    month = SchemaNode(Integer())
    year = SchemaNode(Integer())
    day = SchemaNode(Integer())

class DatesSchema(SequenceSchema):
    date = DateSchema()

class MySchema(MappingSchema):
    name = SchemaNode(String(),
                      description = ’The name of this thing’)
    title = SchemaNode(String(),
                       widget = widget.TextInputWidget(size=40),
                       validator = Length(max=20),
                       description = ’A very short title’)
    password = SchemaNode(String(),
                          widget = widget.CheckedPasswordWidget(),
                          validator = Length(min=5))
    is_cool = SchemaNode(Boolean(),
                         default = True)
    dates = DatesSchema()
    color = SchemaNode(String(),
                       widget = widget.RadioChoiceWidget(values=colors),
                       validator = OneOf((’red’, ’blue’)))

def form_view(request):
    schema = MySchema()
    myform = Form(schema, buttons=(’submit’,))

    if ’submit’ in request.POST:
        controls = request.POST.items()
        try:
            myform.validate(controls)
        except ValidationFailure, e:
            return {’form’:e.render()}
        return {’form’:’OK’}

    return {’form’:myform.render()}

if __name__ == ’__main__’:
    settings = dict(reload_templates=True)
    config = Configurator(settings=settings)
    config.add_view(form_view, renderer=os.path.join(here, 'form.pt'))
    config.add_static_view('static', 'deform:static')
    app = config.make_wsgi_app()
    serve(app)
```

Here's the Chameleon ZPT template named `form.pt`, placed in the same directory:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>
  Deform Sample Form App
</title>
<!-- Meta Tags -->
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<!-- JavaScript -->
<script type="text/javascript" src="static/scripts/deform.js"></script>
<!-- CSS -->
<link rel="stylesheet" href="static/css/form.css" type="text/css" />
</head>
<body id="public">
<div id="container">
<h1>Sample Form</h1>
<span tal:replace="structure form"/>
</div>
</body>
</html>
```

1.9 Using Ajax Forms

To create a form object that uses AJAX, we do this:

```
from deform import Form
myform = Form(schema, buttons=('submit',), use_ajax=True)
```

Creating a Form Object indicates how to create a Form object based on a schema and some buttons. Creating an AJAX form uses the same constructor as creating a non-AJAX form: the only difference between the example provided in the *Creating a Form Object* section and the example above of creating an AJAX form is the additional `use_ajax=True` argument passed to the Form constructor.

If `use_ajax` is passed as `True` to the constructor of a `deform.Form` object, the form page is rendered in such a way that when a submit button is pressed, the page is not reloaded. Instead, the form is posted, and the result of the post replaces the form element's DOM node.

Examples of using the AJAX facilities in Deform are showcased on the http://deformdemo.repoze.org demonstration website:

- Redirection on validation success
- No redirection on validation success

Note that for AJAX forms to work, the `deform.js` and `jquery.form.js` libraries must be included in the rendering of the page that includes the form itself, and the `deform.load()` JavaScript function must be called by the rendering in order to associate the form with AJAX. This is the responsibility of the wrapping page. Both libraries are present in the static directory of the deform package itself. See *Widget Requirements and Resources* for a way to detect which JavaScript libraries are required for a particular form rendering.

1.10 Internationalization

Deform is fully internationalizable and localizable. *gettext* .mo. files exist in the deform and colander packages which contain (currently incomplete) translations to various languages for the purpose of rendering localized error messages.

Following should get you started with *i18n* in pyramid:

```
import deform

from pkg_resources import resource_filename
from pyramid.i18n import get_localizer
from pyramid.threadlocal import get_current_request


def main(global_config, **settings):
    config = Configurator(settings=settings)
    config.add_translation_dirs(
        'colander:locale',
        'deform:locale',
    )

    def translator(term):
        return get_localizer(get_current_request()).translate(term)

    deform_template_dir = resource_filename('deform', 'templates/')
    zpt_renderer = deform.ZPTRendererFactory(
        [deform_template_dir],
        translator=translator)
    deform.Form.set_default_renderer(zpt_renderer)
```

See the Internationalization demo for an example of how deform error and status messages can be localized. This demonstration uses the internationalization and localization features of Pyramid to render Deform error messages into *Chameleon* form renderings.

1.11 API Documentation

1.11.1 Form-Related

1.11.2 Type-Related

See also the type- and schema-related documentation in *Colander*.

1.11.3 Exception-Related

See also the exception-related documentation in *Colander*.

1.11.4 Template-Related

`default_renderer`
The default ZPT template *renderer* (uses the `deform/templates/` directory as a template source).

1.11.5 Widget-Related

`default_resource_registry`
The default *resource registry* (maps Deform-internal *widget requirement* strings to resource paths). This resource registry is used by forms which do not specify their own as a constructor argument, unless `deform.Field.set_default_resource_registry()` is used to change the default resource registry.

1.12 Interfaces

The below are abstract interfaces expected to be fulfilled by various Deform implementations.

1.13 Glossary

appstruct A raw application data structure (complex Python objects).

Chameleon chameleon is an attribute language template compiler which supports the ZPT (Zope Page Templates) templating specification. It is written and maintained by Malthe Borch.

Colander A schema package used by Deform to provide serialization and validation facilities.

cstruct Data serialized by *Colander* to a representation suitable for consumption by the `serialize` method of a `deform` widget, usually while a form is being rendered.

default renderer The template *renderer* used when no other renderer is specified. It uses the *Chameleon* templating engine.

field An object in the graph generated by `deform` that has access to a *schema* node object and a *widget* object. The scope of a field object is generally limited to the scope of a single HTTP request, so field objects are often used to maintain state information during the request.

form controls A sequence of browser renderings of user interface elements. These are also known as "fields" as per the the RFC 2388 definition of "field", however Deform uses the term *field* for another concept, so we call them controls within the Deform documentation.

Gettext The GNU gettext library, used by the `deform` translation machinery.

jQuery jQuery is a JavaScript library for making client side changes to HTML.

JQuery UI A library used by Deform for various widget theming, effects and functionality: See http://jqueryui.com/.

jquery.autocomplete A *jQuery* plugin library that allows for autocompleting a value in a text input, making it easier to find and select a value from a possibly large list. The data may be local or remote. See also http://docs.jquery.com/Plugins/Autocomplete for more details.

jquery.maskedinput A JQuery plugin library that allows for input masks in text inputs. For example, a mask for a US telephone number might be `(999)-999-9999`. See also http://digitalbush.com/projects/masked-input-plugin/. Deform supports input masks in its default `deform.widget.TextInputWidget` widget.

jquery.ui.autocomplete A *JQuery UI* sublibrary for autocompletion of text fields. See http://docs.jquery.com/UI/Autocomplete.

JSON `JSON` (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. See also http://www.json.org/.

Peppercorn A package used by Deform for strutured form submission value deserialization.

pstruct Data deserialized by *Peppercorn* from one or more form controls to a representation suitable for consumption by the `deserialize` method of a `deform` widget, usually while a form is being submitted.

renderer A callable with the signature `(template_name, **kw)` which is capable of rendering a template for use in a deform widget.

renderer A function which accepts a logical template name and a set of keywords, and which returns the rendering of a widget template.

Resource registry An attribute of a Deform form which maps *widget requirement* declarations made by widgets to relative file paths. Useful to obtain all the CSS and/or Javascript resources required by all the widgets in a concrete form rendering. See also *The (High-Level) deform.Field.get_widget_resources() Method*.

schema A nested collection of *schema node* objects representing an arrangement of data.

schema node A schema node can serialize an *appstruct* to a *cstruct* and deserialize a *cstruct* to an *appstruct* (object derived from `colander.SchemaNode` or one of the colander Schema classes). Schemas are a concept used by Deform, but actually implemented and offered by the *Colander* package.

Sequence A widget which allows you to add multiple subwidgets, each of the same type.

TinyMCE Editor TinyMCE is a platform independent web based Javascript HTML WYSIWYG editor control released as Open Source under LGPL by Moxiecode Systems AB. It has the ability to convert HTML TEXTAREA fields or other HTML elements to editor instances. TinyMCE is very easy to integrate into other Content Management Systems.

validator A *Colander* validator callable. Accepts a `node` object and a `value` and either raises an `colander.Invalid` exception or returns `None`. Used in deform as the `validator=` argument to a schema node, ensuring that the input meets the requirements of the schema.

WebOb WebOb is a WSGI request/response library created by Ian Bicking.

widget Serializes a *cstruct* into a form rendering and deserializes a *pstruct* into a *cstruct*.

Widget requirement A sequence of tuples attached to a widget object representing the *logical* Javascript and/or CSS requirements of the widget. See also *Specifying Widget Requirements*.

xhr `xhr` an XMLHTTPRequest. See also http://www.w3.org/TR/XMLHttpRequest/.

1.14 Next Release

1.14.1 Features

- `deform.widget.RichTextWidget` now accepts a dict/two-tuple `options` for specifying arbitrary options to pass to TinyMCE's `init` function. All default options are now part of the class itself (where possible) and can be customised by using `options`. [davidjb]
- `deform.field.Field` now renders a `css_class` on its fieldset, which is set on the schema. This works in the same way as setting a schemas the title inside the fieldset.

1.14.2 Bug Fixes

- Trigger a change event when adding/removing sequence items.

- Add optional label to checkbox widget.
- Make setup_requires depend once again on setuptools_git.
- Raise a ValueError exception when the prototype for a field in a sequence has no name. See https://github.com/Pylons/deform/issues/149

1.15 0.9.7 (2013-03-06)

1.15.1 Bug Fixes

- Readonly checkbox template had a logic error.

1.15.2 Documentation

- Corrected the expected server response when using the Autocomplete widget.

1.16 0.9.6 (2013-01-10)

1.16.1 Bug Fixes

- Fixed remove bug in nested sequences. See https://github.com/Pylons/deform/pull/89
- Fixed bug wherein items added to a sequence nor the initial items rendered in a sequence would not reflect the correct defaults of the item widget. See https://github.com/Pylons/deform/pull/79
- Fix bug where native datetime/date widget rendering competed with jQuery datetime/date widget rendering. See https://github.com/Pylons/deform/pull/142

1.16.2 Dependencies

- Depend on and use zope.deprecation to deprecate Set class.
- Deform now depends on Colander >= 1.0a1 (previously it depended on >= 0.8). It requires Colander 1.0a1's newer `cstruct_children` and `appstruct_children` methods of schema objects as well as being able to import objects from Colander that don't exist in earlier versions.
- Deform now depends on Chameleon >= 2.5.1 (previously it depended on >= 1.2.3). It requires the Markup class supplied by this version or better.
- Deform no longer has a setup_requires dependency on setuptools_git (useless, as the version on PyPI is broken).
- Setup.py now includes all testing requirements in tests_require that are in testing extras and vice versa.

1.16.3 Features

- Allow SelectWidget to produce <optgroup> HTML tags. See https://github.com/Pylons/deform/pull/87
- Allow `deform.form.Form` constructor to accept an `autocomplete` keyword argument, which controls the `autocomplete` attribute of the form tag.
- Add Python 3.3 Trove classifier.

- Pass through unknown keys in a `filedict` FileData serialization (FBO of passing out of band information).
- `deform.Set` type deprecated in favor of use of `colander.Set`.
- Give the preview_url method of the tempstore access to the stored item. [tomster]
- Add `style` attribute/arguments to textinput-related widgets allowing you to set the style of the tag by hand.
- Allow `deform.widget.SequenceWidget` constructor to accept an `orderable` keyword argument. Default is `False`. If `True`, allow drag-and-drop reordering of SequenceWidget items (via jQuery UI Sortable).
- The default widget for the colander.Money type is now deform.widgets.MoneyInputWidget.
- Built-in widgets may have a 'readonly' attribute/constructor-argument, to indicate that a form field associated with the widget should use its readonly template instead of its normal readwrite template. A `readonly` keyword argument can still be passed to `Field.serialize` to render a field as readonly, like in older versions.
- `deform.field.Field` now has a `__contains__` method, which returns `True` if the named field is a subfield of the field on which it is called.
- `deform.field.Field` now has a `validate_pstruct` method which works like `validate` except it accepts a pstruct directly instead of accepting a list of controls.
- `deform.field.Field.validate` now accepts a `subcontrol` argument for validating a submapping of a form.
- In support of "retail" form rendering, the `serialize` method of widgets now accepts arbitrary keyword arguments. These are used as top-level value overrides to widget templates.
- In support of "retail" form rendering, the `serialize` method of a Field now accepts arbitrary keyword arguments. These are passed along to it's widget's `serialize` method.
- It is now possible to pass an `appstruct` argument to the `deform.Field` (and by extension, the `deform.Form`) constructor. When you do so, you can omit passing an `appstruct` argument to the `render` method of the field/form. Fields set a cstruct value recursively when supplied with an `appstruct` argument to their constructor. This is in support of "retail" form rendering.
- Form/field objects are now initialized with a cstruct (recursively) when created. This means that accessing form.cstruct will return the current set of rendering values. This value is reset during validation, so after a validation is done you can re-render the form to show validation errors. This is in support of "retail" form rendering.
- Form/field objects now have peppercorn-field-outputting methods: `start_mapping`, `end_mapping`, `start_sequence`, `end_sequence`, `start_rename`, `end_rename` in support of retail form rendering.
- The `deform.Field` (and therefore `deform.Form`) classes now expose a `render_template` method, which injects `field` and `cstruct` into the dictionary passed to the template if they don't already exist in the `**kw` passed. This is in support of retail form rendering.
- Add `set_appstruct` and `set_pstruct` methods to Field; these accept, respectively, an appstruct or a pstruct and set the cstruct related to the field to the deserialized or serialized value.

1.16.4 Documentation

- Add a (weak) "Retail Form Rendering" chapter to the docs.

1.17 0.9.5 (2012-04-27)

- Add translations for TinyMCE. Thanks OCHIAI, Gouji.

- Japanese translation thanks to OCHIAI, Gouji.
- Modified Russian translation thanks to aleksandr.rakov
- Date(Time)Widget supports now options to configure it, thx to gaston tjebbes, kiorky
- FileUploadWidget now sanitizes IE/Windows whole-path filenames before passing them back to the caller during deserialization/validation.
- Add docs and dev setup.py aliases ala Pyramid.
- Add MoneyInputWidget widget type.
- Allow a custom i18n domain to be used for the "Add ${subitem_title}" link of a SequenceWidget. See https://github.com/Pylons/deform/issues/85 .
- Allow the use of Integer values with SelectWidget. See https://github.com/Pylons/deform/issues/81 .
- CheckedInputWidget and CheckedPasswordWidget now populate the "confirm" element with the cstruct value (for edit forms).
- Update to JQuery 1.7.2.
- Update to jquery.form 3.09.

1.18 0.9.4 (2012-02-14)

- No longer Python 2.5 compatible. Python 2.6+ is required.
- Python 3.2 compatible.
- Translate title attribute for remove button in sequence fields.
- Do not output empty error messages for sequence items. After translation these would insert the PO file metadata.
- Update to lingua for translations, add french translation
- fix multiple i18n issues.
- Fix a bug where displaying error could lead on an error when you have imbricated Mapping objects
- Fix issue #54: form.pt does not show validation errors from the top node of the schema. See https://github.com/Pylons/deform/issues/54 for more information.
- Previously, all CheckedInputWidget and CheckedPasswordWidget fields had hardcoded input[name] attributes of 'value' and 'confirm'. When deserializing a form, this caused colander.null to be passed to the widget deserialization function since neither submitted value matched the name of the field. This change simply replaces 'value' with the name of the field and 'confirm' with the name of the field with '-confirm' appended.
- In select widget, add css_class to <select> rather than only <option>.
- Allow RichText fields to load their editor only after clicking on them
- There is no longer a `deform_ajaxify` global javascript function. Instead forms are AJAXified directly by the javascript callback for the form.

1.19 0.9.3 (2011-08-10)

- Update Dutch translations.
- Translate title and description of items for sequence fields.

- Add a new API method to field objects: `translate`. This method will use the translator passed to the underlying renderer to translate message ids into text.

1.20 0.9.2 (2011-07-22)

- Chameleon 2 compatibility.
- Use default widgets for a schema's baseclass if known instead of always falling back to a text widget.
- Deform now includes a `beautify.css` (contributed by Ergo^) in its static directory, which can be used to make form element styling prettier.
- Moved `deformdemo` into its own package and Github repository (https://github.com/Pylons/deformdemo).

1.21 0.9.1 (2011-06-23)

- Add Dutch translation.
- Add the `deform.widget.DateTimeWidget` widget, which uses the jQueryUI Timepicker add-on.

  `DateTimeWidget` uses the ISO8601 combined date and time format internally, as expected by `colander.DateTime`, but converts to the more friendly separate date and time format for display in the widget.

  This widget is now the default for colander.DateTime schema types.
- Upgrade to jquery-ui 1.8.11, as required by the timepicker.
- Compile all `.po` files to `.mo` in `deform/locale` and remove Texan locale (funny, but breaks `python setup.py compile_catalog` with an UnknownLocale error.)
- Fix references to repoze.bfg and update obsoletes URLs in the demo application
- Remove unused `jquery.autocomplete.min.js` file from static directory.
- SelectWidget now has a `size` attribute to support single select widgets that are not dropdowns.
- The value fed to the `deform.form.Button` class as `name` would generate an invalid HTML id if it contained spaces. Now it converts spaces to underscores if they exist in the name. See https://github.com/Pylons/deform/pull/14 .
- Deformdemo application now has a Time field demonstration.
- Deform Chameleon templates now contain i18n:translate tags.
- German translation updated.
- Fixed invalid HTML generated for "select" widget.
- When using an ajax form without a redirect, a submit overwrites the form. In the case of a form validation failure on first submit, no event handlers were registered to submit the form via ajax on the second submit. This is now fixed. See https://github.com/Pylons/deform/pull/1 .

1.22 0.9 (2011-03-01)

- Moved to GitHub (https://github.com/Pylons/deform).
- Added tox.ini for testing purposes.

- Fix select dropdown behavior on Firefox by fixing CSS (closes http://bugs.repoze.org/issue152).
- Removed `wufoo.css`, minimized `form.css`. Changed templates around to deal with CSS changes.
- Sequence widgets now accept a min_len and a max_len argument, which influences its display of close and add buttons.
- Convert demo application from repoze.bfg to Pyramid.
- Depend on Chameleon<1.999 (deform doesn't yet work with Chameleon 2).

1.23 0.8.1 (2010-12-17)

1.23.1 Features

- Allow `deform.form.Button` class to be passed a `disabled` flag (false by default). If a Button is disabled, its HTML `disabled` setting will be set true.

1.24 0.8 (2010-12-02)

1.24.1 Features

- Added Polish locale data: thanks to Marcin Lulek.

1.24.2 Bug Fixes

- Fix dynamic sequence item adding on Chrome and Firefox 4. Previously if there was a validation error rendering a set of sequence items, the "add more" link would be rendered outside the form, which would cause it to not work. Wrapping the sequence item <li> element in a <ul> fixed this.

1.25 0.7 (2010-10-10)

1.25.1 Features

- Added Danish locale.
- Added Spanish locale: thanks to David Cerna for the translations!
- `DatePartsWidget` now renders error "Required" if all blank or "Incomplete" if partially blank for consistency with the other widgets.
- Different styling involving <li> and <ul> for checkbox choice, checked input, radio choice, checked password, and dateparts widgets (via Ergo^). See http://bugs.repoze.org/issue165.

1.25.2 Dependencies

- Deform now depends on `colander` version 0.8 or better (the demo wants to use schema bindings).
- Deform now depends on `Chameleon` (uppercase) rather than `chameleon` to allow for non-PyPI servers.

1.25.3 Demo

- New addition to the demonstration application: schema binding.

1.26 0.6 (2010-09-03)

1.26.1 Features

- Sequence widgets are no longer `structural` by default; they now print the label of the sequence above the sequence adder.
- Radio buttons in a radio button choice widget are now spaced closer together and the button is on the left hand side.
- The sequence remove button is no longer an image.
- The sequence widget now puts the sequence adding link *after* any existing items in the sequence (previously the link was always beneath the sequence title).
- It is now possible to associate a widget with a schema node within the schema directly. For example:

  ```
  import colander
  import deform.widget

  class MySchema(Schema):
      description = colander.SchemaNode(
                          colander.String(),
                          widget=deform.widget.RichTextWidget()
                          )
  ```

  For more information, see "Changing the Default Widget Associated With a Field" in the documentation.
- The constructor of `deform.Field` objects (and thus `deform.Form` objects) now accept arbitrary keyword arguments, each of which is attached to the field or form object being constructed, as well as being attached to its descendant fields recursively.
- The form object's template now respects the `css_class` argument / attribute of the form node.
- CheckboxChoice and RadioChoice widgets now use <ul> and <li> to display individual choice elements (thanks to Ergo^), and both widgets put the label after the element instead of before as previously.
- The `deform.widget.AutocompleteInputWidget` widget now uses *JQuery UI's autocomplete sublibrary <http://docs.jquery.com/UI/Autocomplete>* instead of the `jquery.autocomplete` library to perform its job in order to reduce the number of libraries needed by Deform. Some options have been changed as a result, and the set of resources returned by `form.get_widget_resources` has changed.

  This change also implies that when a widget which uses a remote URL as a `values` input, the remote URL must return a JSON structure instead of a `\n`-delimited list of values.

1.26.2 Requirements

- This Deform version requires `colander` version 0.7.3 or better.

1.26.3 Bug Fixes

- `RichTextWidget`, `AutocompleteInputWidget`, `TextInputWidget` with input masks, and `CheckedInputWidget` with input masks could not be used properly within sequences. Now they can be. See also `Internal` and `Backwards Incompatibilities` within this release's notes. This necessitated new required `deform.load()` and `deform.addCallback()` JavaScript APIs.
- Radio choice widgets included within a submapping no longer put their selections on separate lines.
- Rich text widgets are now 500 pixels wide by default instead of 640.
- RadioChoiceWidgets did not work when they were used within sequences. Making them work required some changes to the its template and it added a dependency on `peppercorn` >= 0.3.
- To make radio choice widgets work within sequences, the deform.addSequenceItem JavaScript method needed to be changed. It will now change the value of `name` attributes which contain a marker that looks like an field oid (e.g. `deformField1`), and, like the code which changes ids in the same manner, appends a random component (e.g. `deformField1-HL6sgP`). This is to support radio button groupings.
- The mapping and sequence item templates now correctly display errors with `msg` values that are lists. Previously, a repr of a Python list was displayed when a widget had an error with a `msg` value that was a list; now multiple <p> nodes are inserted into the rendering, each <p> node containing an individual error message. (Note that this change requires colander 0.7.3).

1.26.4 Backwards Incompatibilities

- The JavaScript function `deform.load()` now *must* be called by the HTML page (usually in a script tag near the end of the page, ala `<script..>deform.load()</script>`) which renders a Deform form in order for widgets which use JavaScript to do proper event and behavior binding. If this function is not called, built-in widgets which use JavaScript will no longer function properly.
- The JavaScript function `deformFocusFirstInput` was removed. This is now implied by `deform.load()`.
- The `closebutton_url` argument to the SequenceWidget no longer does anything. Style the widget template via CSS to add an image.

1.26.5 Internal

- Provided better instructions for running the demo app and running the tests for the demo app in `deformdemo/README.txt`.
- Try to prevent false test failures by injecting sleep statements in things that use `browser.key_press`.
- Moved `deformdemo/tests/test_demo.py` to `deformdemo/test.py` as well as moving `deformdemo/tests/selenium.py` to `deformdemo/selenium.py`. Removed the `deformdemo/tests` subdirectory.
- The date input widget now uses JQueryUI's `datepicker` functionality rather than relying on JQuery Tools' `date` input. The latter was broken for sequences, and the former works fine.
- The various deform* JavaScript functions in `deform.js` have now been moved into a top-level namespace. For example, where it was necessary to call `deformFocusFirstInput()` before, it is now necessary to call `deform.focusFirstInput()`.
- Make the TinyMCE rich text widget use `mode: 'exact'` instead of `mode: 'textareas'`.

- richtext, autocomplete_input, textinput, checked_input, and dateinput, and form templates now use the new deform.addCallback indirection instead of each registering their own JQuery callback or performing their own initialization logic, so that each may be used properly within sequences.
- Change sequence adding logic to be slightly simpler.
- The sample app form page now calls deform.load() rather than deformFocusFirstInput().
- Added new demo app views for showing a sequence of autocompletes, a sequence of dateinputs, a sequence of richtext fields, a sequence of radio choice widgets and a sequence of text inputs with masks and tests for same.

1.26.6 Documentation

- Added a note about get_widget_resources to the "Basics" chapter.
- Added a note about deform.load() JavaScript requiredness to the "Basics" chapter.
- Add new top-level sections named Widget Templates and Widget JavaScript to the "Widgets" chapter.

1.27 0.5 (2010-08-25)

1.27.1 Features

- Added features which make it possible to inquire about which resources (JavaScript and CSS resources) are required by all the widgets that make up a form rendering. Also make it possible for a newly created widget to specify its requirements. See "Widget Requirements and Resources" in the widgets chapter of the documentation.
- Add the get_widget_requirements method to deform.Field objects.
- Add the get_widget_resources method to deform.Field objects.
- Allow deform.Field (and deform.Form) objects to accept a "resource registry" as a constructor argument.
- Add the deform.Field.set_widgets method, which allows a (potentially nested) set of widgets to be applied to children fields of the field upon which it is called.
- Add the deform.widget.TextInputCSV widget. This widget is exactly like the deform.widget.TextAreaCSV widget except it accepts a single line of input only.
- The default widget for colander.Tuple schema types is now deform.widget.TextInputCSV.
- The deform.widget.FileUploadWidget now returns an instance of deform.widget.filedict instead of a plain dictionary to make it possible (using isinstance) to tell the difference between file upload data and a plain data dictionary for highly generalized persistence code.

1.28 0.4 (2010-08-22)

1.28.1 Bug Fixes

- When the hidden widget is used to deserialize a field, return colander.null rather than the empty string so that it may be used to represent non-text fields such as colander.Integer. This is isomorphic to the change done previously to deform.TextInputWidget to support nontextual empty fields.

- Fix typo about overriding templates using set_zpt_renderer in templating chapter.
- Fix link to imperative schema within in Colander docs within "Basics".
- Remove duplicate `deform.widget.DateInputWidget` class definition.

1.28.2 Features

- Add a `deform.widget.RichTextWidget` widget, which adds the TinyMCE WYSIWIG javascript editor to a text area.
- Add a `deform.widget.AutocompleteInputWidget` widget, which adds a text input that can be supplied a URL or iterable of choices to ease the search and selection of a finite set of choices.
- The `deform.widget.Widget` class now accepts an extra keyword argument in its constructor: `css_class`.
- All widgets now inherit a `css_class` attribute from the base `deform.widget.Widget` class. If *css_class*' contains a value, the "primary" element in the rendered widget will get a CSS `class` attribute equal to the value ("primary" is defined by the widget template's implementor).
- The `deform.Field` class now as an __`iter`__ method which iterates over the children fields of the field upon which it is called (`for item in field` == `for item in field.children`).

1.29 0.3 (2010-06-09)

1.29.1 Bug Fixes

- Change default form action to the empty string (rather than `.`). Thanks to Kiran.

1.29.2 Features

- Add `deform.widget.DateInputWidget` widget, which is a date picker widget. This has now become the default widget for the `colander.Date` schema type, preferred to the date parts widget.
- Add text input mask capability to `deform.widget.TextInputWidget`.
- Add text input mask capability to `deform.widget.CheckedInputWidget`.

1.29.3 Backwards Incompatibilities

- Custom widgets must now check for `colander.null` rather than `None` as the null sentinel value.
- Dependency on a new (0.7) version of Colander, which has been changed to make using proper defaults possible; if you've used the `default` argument to a `colander.SchemaNode`, or if you've defined a custom Colander type, you'll want to read the updated Colander documentation (particularly the changelist). Short story: use the `missing` argument instead.
- If you've created a custom widget, you will need to tweak it slightly to handle the value `colander.null` as input to both `serialize` and `deserialize`. See the Deform docs at http://docs.repoze.org/deform for more information.

1.30 0.2 (2010-05-13)

- Every form has a formid now, defaulting to `deform`. The formid is used to compute the id of the form tag as well as the button ids in the form. Previously, if a formid was not passed to the Form constructor, no id would be given to the rendered form and the form's buttons would not be prefixed with any formid.
- The `deform.Form` class now accepts two extra keyword arguments in its constructor: `use_ajax` and `ajax_options`.

  If `use_ajax` is `True`, the page is not reloaded when a submit button is pressed. Instead, the form is posted, and the result replaces the DOM node on the page.

  `ajax_options` is a string which allows you to pass extra options to the underlying AJAX form machinery when `use_ajax` is True.
- Added a couple Ajax examples to the demo app.
- Add a rudimentary Ajax chapter to the docs.

1.31 0.1 (2010-05-09)

- Initial release.

CHAPTER

TWO

DEMONSTRATION SITE

Visit deformdemo.repoze.org to view an application which demonstrates most of Deform's features. The source code for this application is also available in the deform package on GitHub.

CHAPTER

THREE

SUPPORT AND DEVELOPMENT

To report bugs, use the bug tracker.

If you've got questions that aren't answered by this documentation, contact the Pylons-discuss maillist or join the

#pylons IRC channel `irc://irc.freenode.net/#pylons`.

Browse and check out tagged and trunk versions of `deform` via the deform package on GitHub. To check out the trunk, use this command:

```
git clone git://github.com/Pylons/deform.git
```

To find out how to become a contributor to `deform`, please see the Pylons Project contributor documentation.

CHAPTER

FOUR

INDEX AND GLOSSARY

- *genindex*
- *modindex*
- *search*

CHAPTER

FIVE

THANKS

Without these people, this software would not exist:

- The Formish guys (http://ish.io)
- Tres Seaver
- Fear Factory (http://fearfactory.com)
- Midlake (http://midlake.net)