Ambidexterity Documentation

Release 1.0

Steve McMahon

Feb 21, 2019

Contents

Ambidexterity provides a through-the-web editor for Plone's Dexterity views and field defaults, validators and vocabularies.

Contents:

CHAPTER 1

Tutorial

Ambidexterity provides through-the-web (TTW) editing of validators, dynamic defaults and vocabularies for Dexterity content types. It also allows you to create custom view templates for presenting those content types.

This tutorial will walk you through the creation of simple validators and dynamic defaults.

1.1 Our scenario

We're going to create a TTW Dexterity content type with two custom fields:

- A string field for a phone number in a standard international format. We'll validate that field using a regular expression test.
- A date field that will automatically default to the current date.

We'll also customize the view template for our content type.

1.2 Preliminary steps

You should install `collective.ambidexterity` by adding it to your buildout and running buildout. Activate it by visiting `Add-ons` control panel in `site setup`.

Note: Ambidexterity currently only works with Plone 5.x.

1.3 Create a test content type and fields

Visit the Dexterity control panel in site setup. Add a new content type named `Test content type` (the name isn't actually important).

To your new content type, add two fields:

- A string field titled *Phone number*; and
- A date field titled *Start date*.

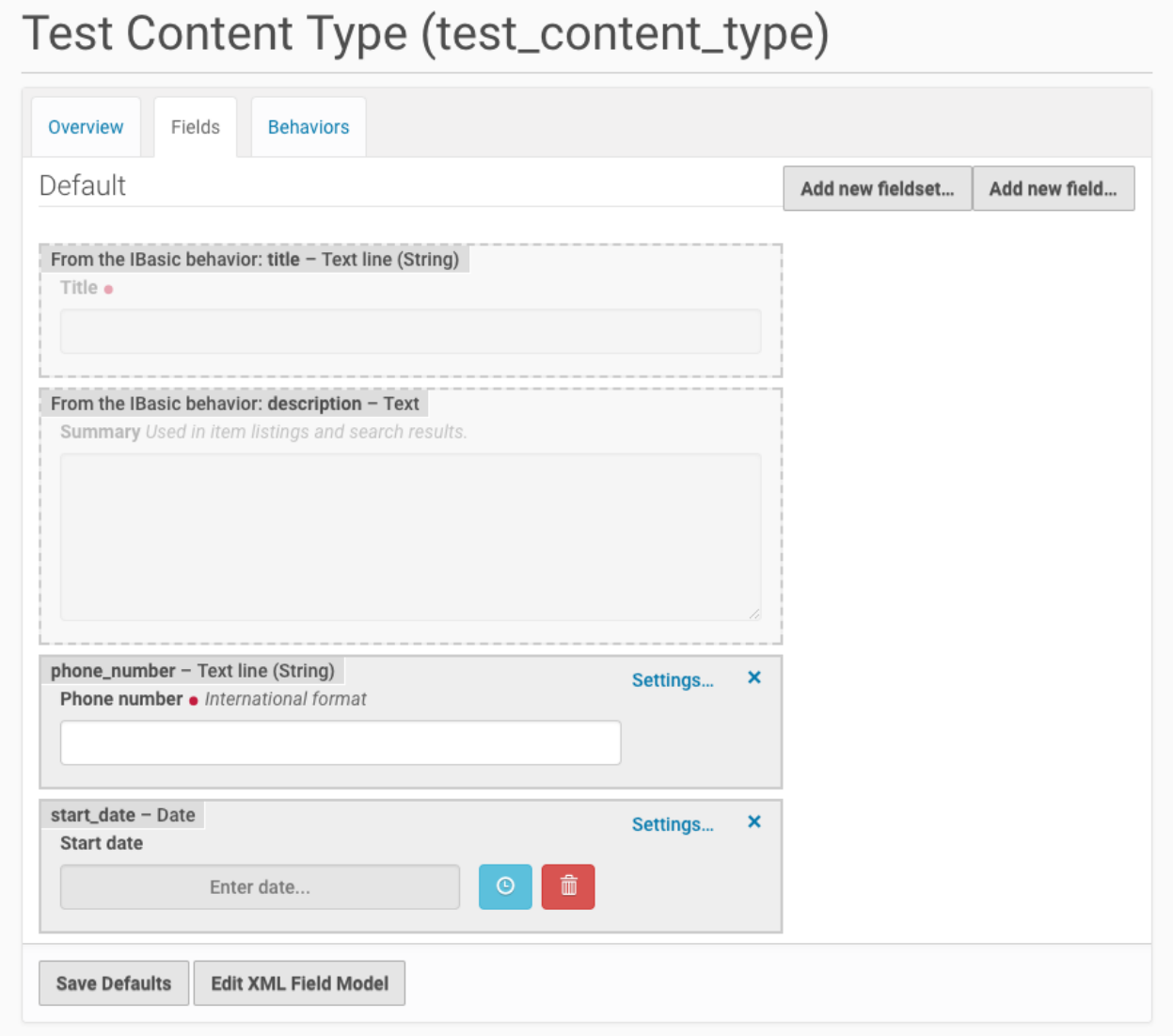

Fig. 1: The Dexterity field editor showing our sample content type and its two test fields.

1.4 The Ambidexterity editor

Return to *Site setup* and look to the bottom of the page for the *Add-on Configuration* section. Select *Ambidexterity*, giving you a view that should look like this:

The *Content types* drop-down field allows you to select the Dexterity content type on which you wish to act. Other options will appear depending on what Ambidexterity can do with the selected content type.

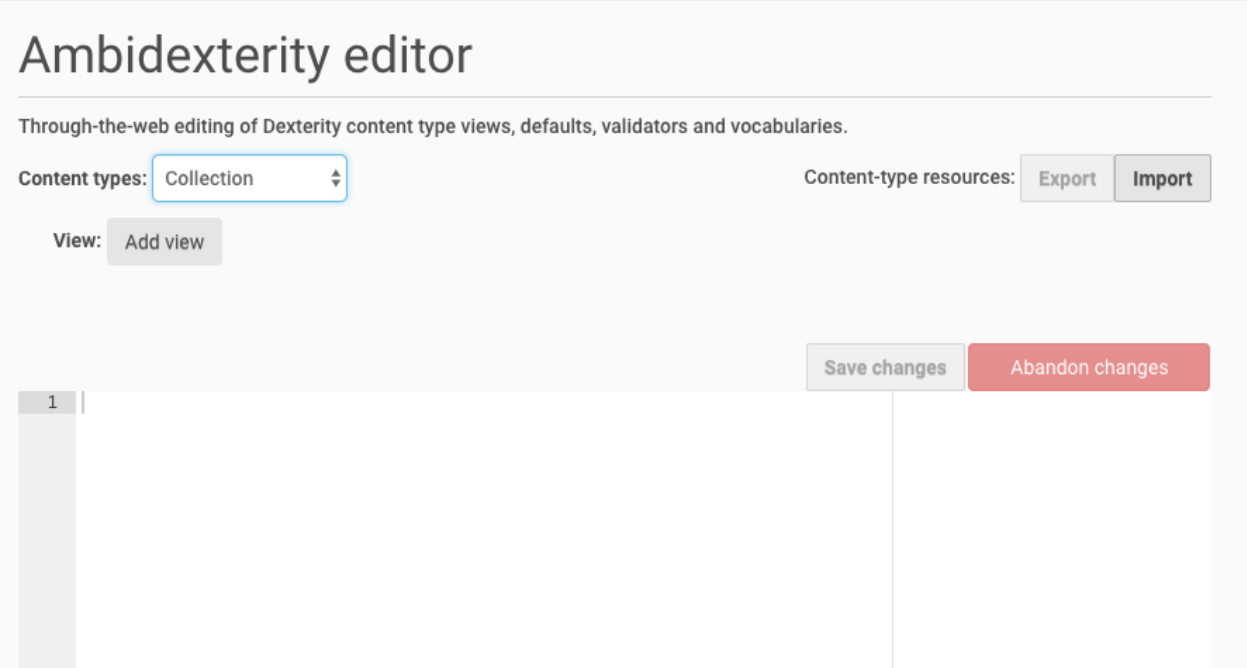

Fig. 2: The Ambidexterity editor. Note that there is only one option for a built-in content type: to create a custom view.

1.4.1 Varieties of content types

For Ambidexterity's purposes, there are two types of content types:

- Those defined through-the-web; and,
- Those defined in Python packages, such as Plone's built-in content types.

For content types defined TTW, we will be able to edit defaults, validators, vocabularies and view templates.

For content types defined in Python packages, we will only be able to edit view templates.

1.4.2 Selecting content types and fields

Use the drop-down *Content types* to select your test content type:

Once you've selected a TTW content type, a *Fields* drop-down list will appear. You may use it to select a field for Ambidexterity editing. Only the fields you added will be available; Ambidexterity does not work with fields added through Dexterity behaviors.

Once you've selected a TTW content type and one of your added fields, lots of new *script* action buttons appear. You'll have the options to add default, validator and vocabulary scripts. (The *Vocabulary* script option only appears for *Choice* and *Multiple Choice* field types.)

1.5 Adding and editing a validator

Select the *Phone number* field. Press the *Add validator* button. The *Add validator* button will be replaced with an *Edit validator* button. A *Remove validator* button appears that you may use to remove the validator script.

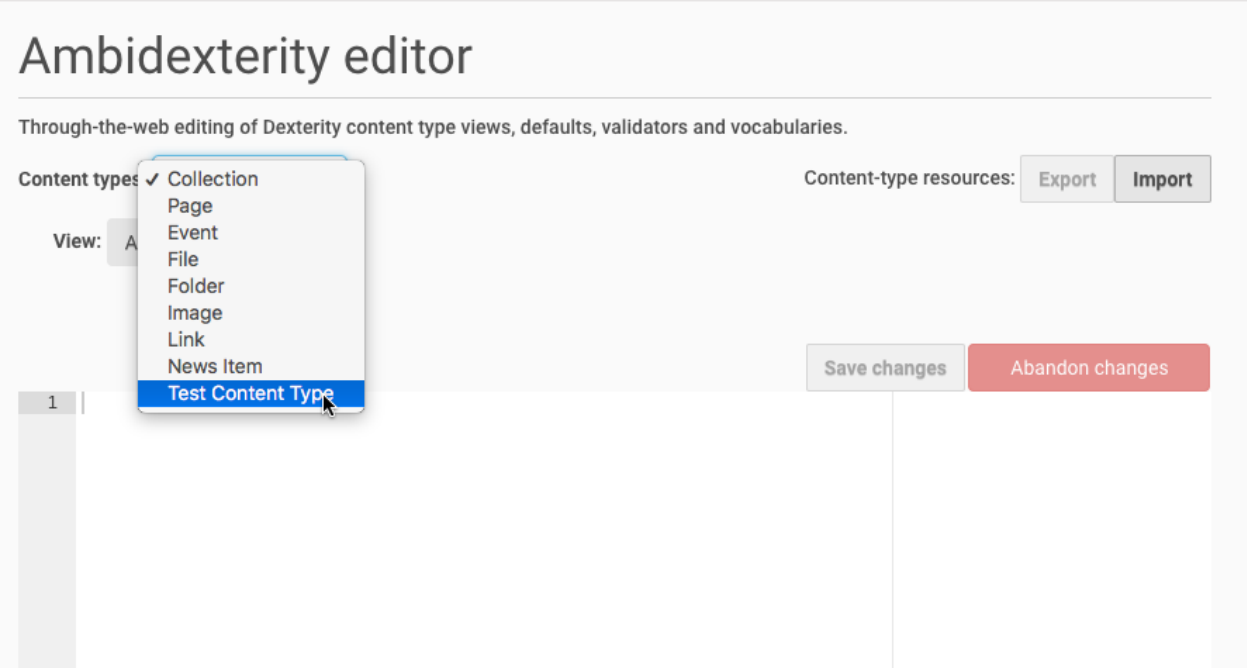

Fig. 3: The Ambidexterity editor: selecting a content type.

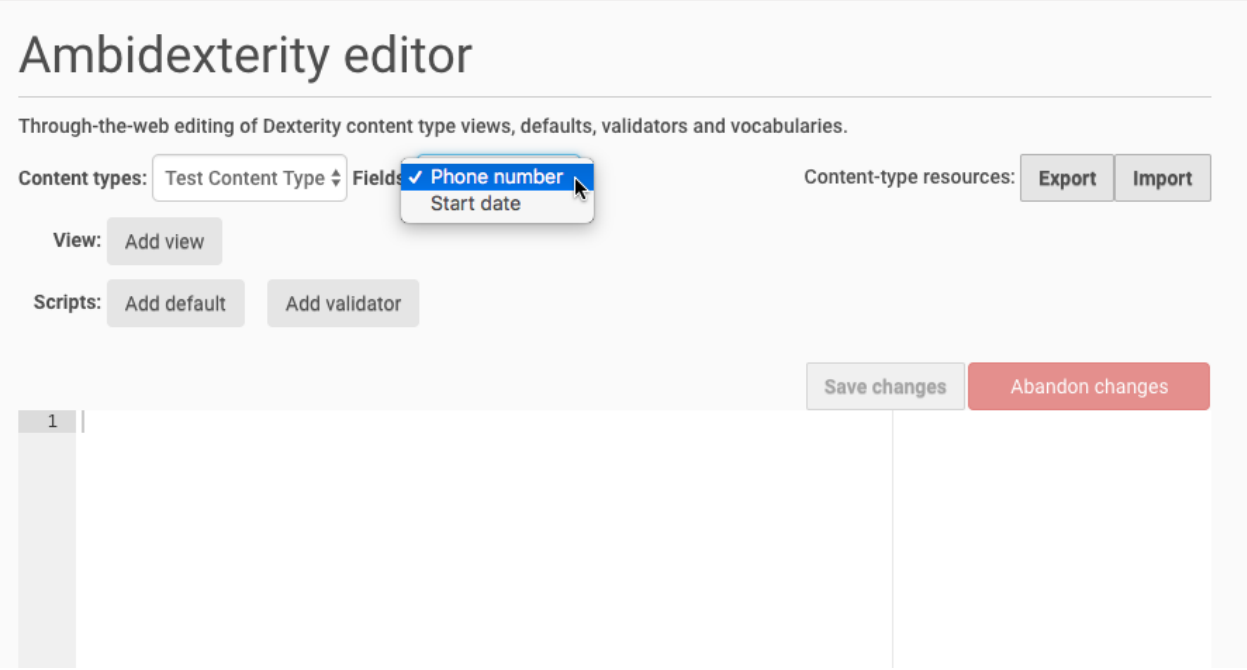

Fig. 4: The Ambidexterity editor: selecting a content type's field. Note that there are new options for our test content type since it was built TTW.

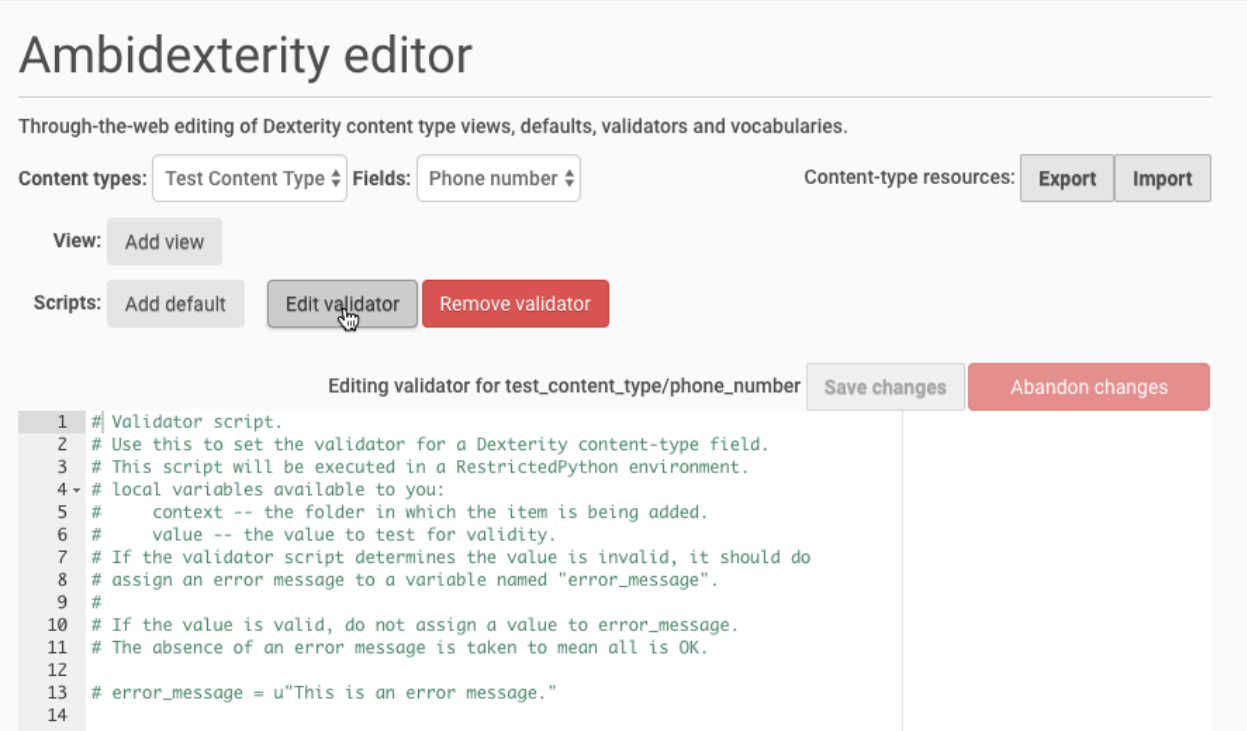

Fig. 5: Editing a validator.

The numbered-lines section of the page is now a text editor and contains the code for a sample validator. While editing, we gain buttons to save or abandon changes.

1.5.1 The workings of a validator script

Note: RestrictedPython

All of our scripts are a Python with some special limitations defined by RestrictedPython.

RestrictedPython is meant to provide a safety net for programmers that are not familiar with the Plone/Zope security model. It limits built-in classes, modules and functions. It also controls object database access, limiting access to items that are available to the current user. The *current user* is not you; if you're using the Ambidexterity editor, you have great powers (and great responsibility). Rather, the *current user* will be the contributor adding or editing the content item.

Your validator script has a special global variable, `value`. That's the field value input by the user. In a validator script, we want to test that value against our expectations. If it fails the test, we want to return an error message.

Look at the last line in the editor:

```
# error_message = u"This is an error message."
```

Change that to read:

```
import re
```

(continues on next page)

(continued from previous page)

```
if re.match(r"^\+(?:[0-9] ?){6,14}[0-9]$", value) is None:
    error_message = u"Phone number must comply with E.164."
```

The `re` *regular expression* module is one of the few that you may import in RestrictedPython. It's particularly useful for validating strings. The `re.match` function tests a regular expression against a string. If the expression matches, a `match` object is returned. If there is no match, Python's `None` is returned.

Our code looks for that `None` value. If it's found, we set a local `error_message` variable to a string. If set, this error message will be displayed on the content item's edit form.

If `error_message` is not set, or is set to `None`, Ambidexterity will interpret that as a sign that the input value is OK.

Now, save it and try it out by adding a test content type item.

1.6 Adding and editing a dynamic default

Return to the Ambidexterity editor. Select your test content type and the *Start date* field. Push the *Add default* button and watch a sample default script fill the editor.

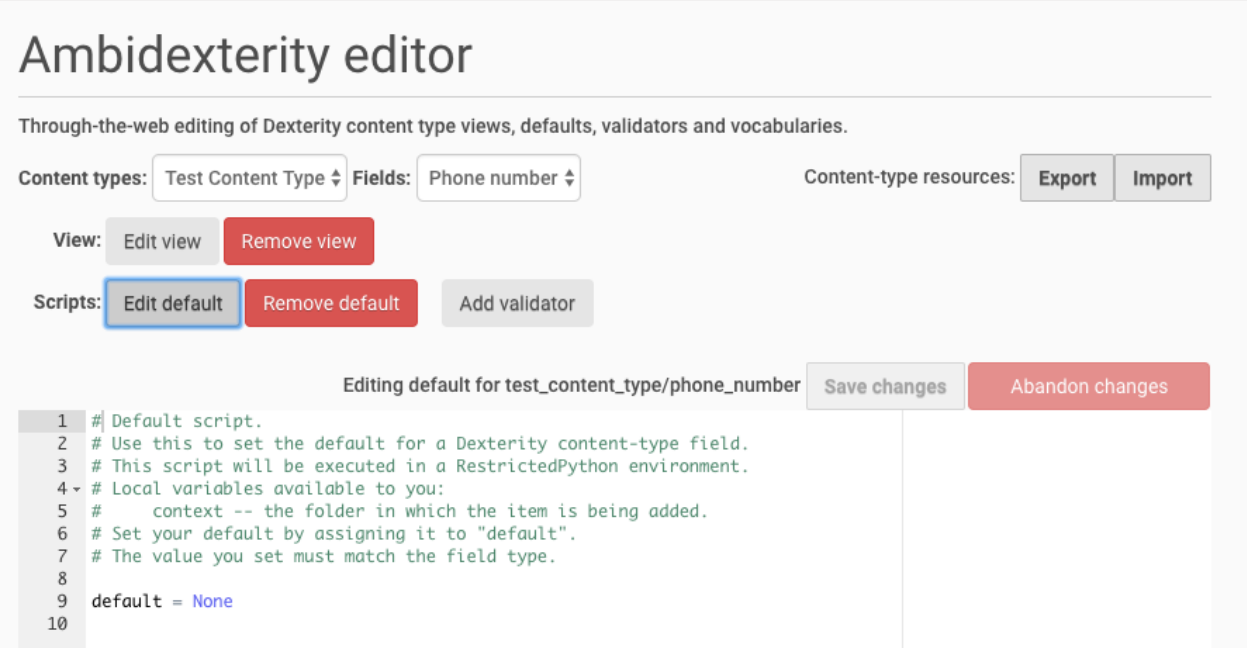

Fig. 6: Editing a dynamic default.

For a default script, we want to set a local `default` variable to the desired value.

Look for the line:

```
default = None
```

Change it to:

```
default = 'Tuesday'
```

Save your changes and try to add a new item for your content type.

1.6.1 Don't fear the Traceback

This isn't what we wanted:

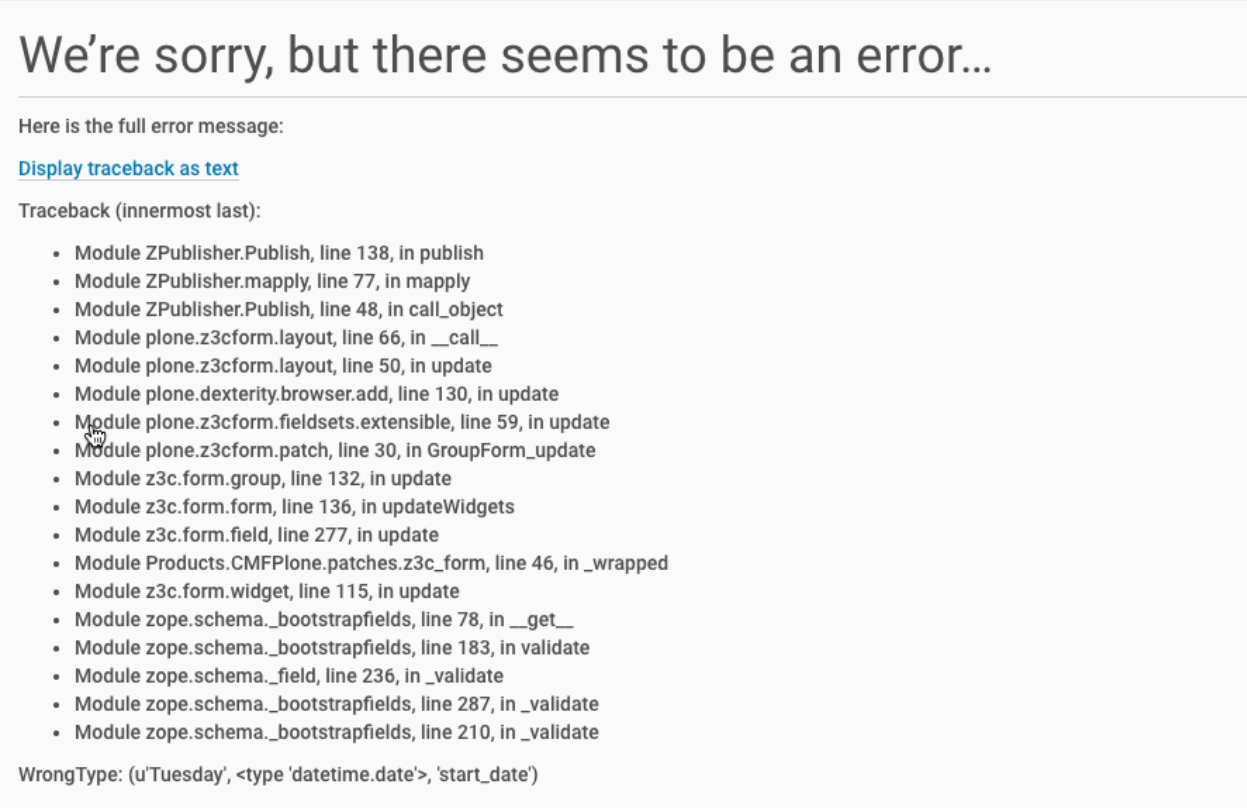

Fig. 7: An error adding a content type with a bad default.

If you've never done Plone programming before, you may have never seen this on a Plone page. It's a standard Python *traceback*. You see it because you're a highly privileged user; a less privileged user would see a message telling them to contact the site administration.

Here's the text of our traceback:

```
2017-10-09 14:35:38 ERROR Zope.SiteErrorLog 1507584938.270.45842617267 http://
↪lumpy:8080/Plone/++add++test_content_type
Traceback (innermost last):
  Module ZPublisher.Publish, line 138, in publish
  Module ZPublisher.mapply, line 77, in mapply
  Module ZPublisher.Publish, line 48, in call_object
  Module plone.z3cform.layout, line 66, in __call__
  Module plone.z3cform.layout, line 50, in update
  Module plone.dexterity.browser.add, line 130, in update
  Module plone.z3cform.fieldsets.extensible, line 59, in update
  Module plone.z3cform.patch, line 30, in GroupForm_update
  Module z3c.form.group, line 132, in update
  Module z3c.form.form, line 136, in updateWidgets
  Module z3c.form.field, line 277, in update
  Module Products.CMFPlone.patches.z3c_form, line 46, in _wrapped
  Module z3c.form.widget, line 115, in update
  Module zope.schema._bootstrapfields, line 78, in __get__
```

(continues on next page)

(continued from previous page)

```
  Module zope.schema._bootstrapfields, line 183, in validate
  Module zope.schema._field, line 236, in _validate
  Module zope.schema._bootstrapfields, line 287, in _validate
  Module zope.schema._bootstrapfields, line 210, in _validate
WrongType: (u'Tuesday', <type 'datetime.date'>, 'start_date')
```

An experienced Python programmer knows how to read a Traceback. If it's relatively new to you, the most important thing to know is to start reading from the bottom. The lines at the top of the traceback belong to Plone; one or more lines at the bottom will belong to you. Start at the bottom and read up until you encounter something you own.

In this case, the key line is the last one. What it's saying is pretty clear: a `datetime.date` was expected. We tried to assign a string, *Tuesday*, when we should have provided a `datetime.date`.

The lesson to learn here is that the *default* you provide must be of a Python type that matches the field type. Date fields must receive dates (datetime.date), DateTime fields must receive `datetime.datetime`, integer fields must receive integers.

Return to the Ambidexterity editor, and we'll fix this.

Change your code to read:

```
from datetime import date

default = date.today()
```

`datetime` is another module allowed in RestrictedPython. `datetime.date.today()` returns the current system date. We know that because *datetime* is a standard Python module, with full documentation in any handy Python reference.

Save your changes. Try again adding a content type. This one should work.

1.7 Adding and editing a view template

If you've successfully added a test content item, the current view of the time should look something like:

Let's change that! Return to the Ambidexterity editor; select your content type; push the *Add view* button. As with scripts, you'll see the *add* button replaced with a *view* button and a new *remove* button:

The code you're looking at is a Zope Page Template (ZPT). It's standard XML with a few extra XML name spaces. The `TAL` namespace is for *template attribute language* and provides mechanisms for inserting and testing dynamic content. The `METAL` namespace is for ZPT macros and allows us to make use of a master page template, only changing the content area.

ZPT is well-documented in it's Reference. The Plone page templates reference covers its use with Plone, including the workings of Plone's master page template.

Let's make a simple change. Look for the core of the content:

```
<p>
  This is the default Ambidexterity view for <span tal:replace="context/portal_type">
↪portal type</span>.
</p>
```

and replace it with:

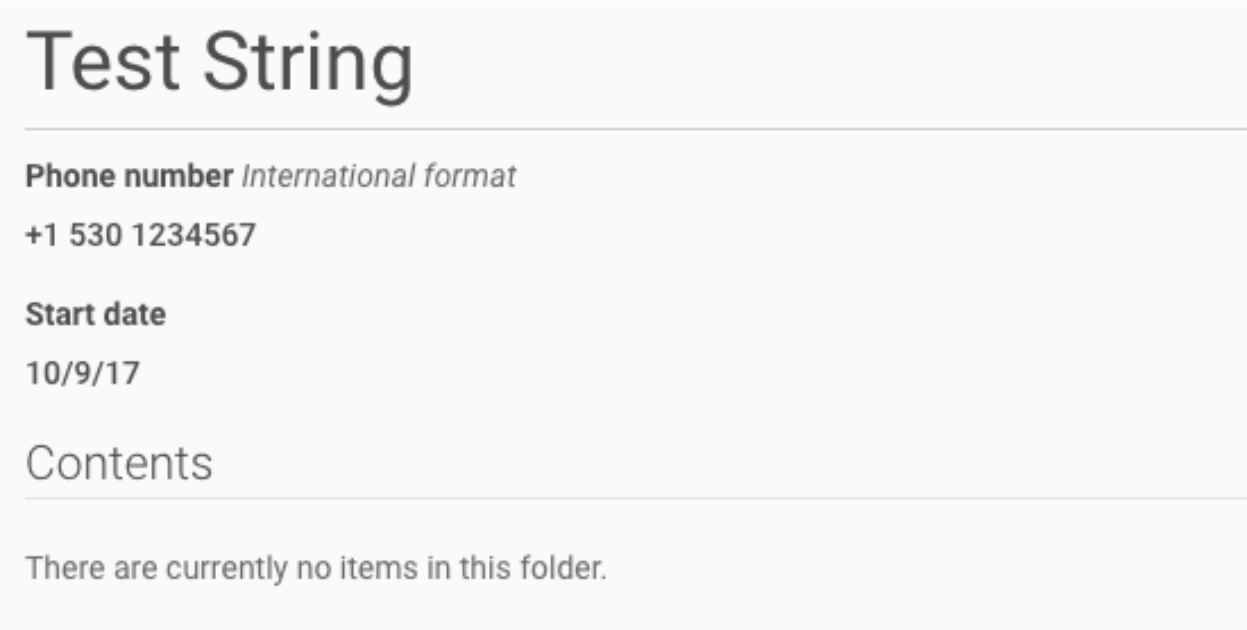

Fig. 8: Dexterity's default view for our content type.

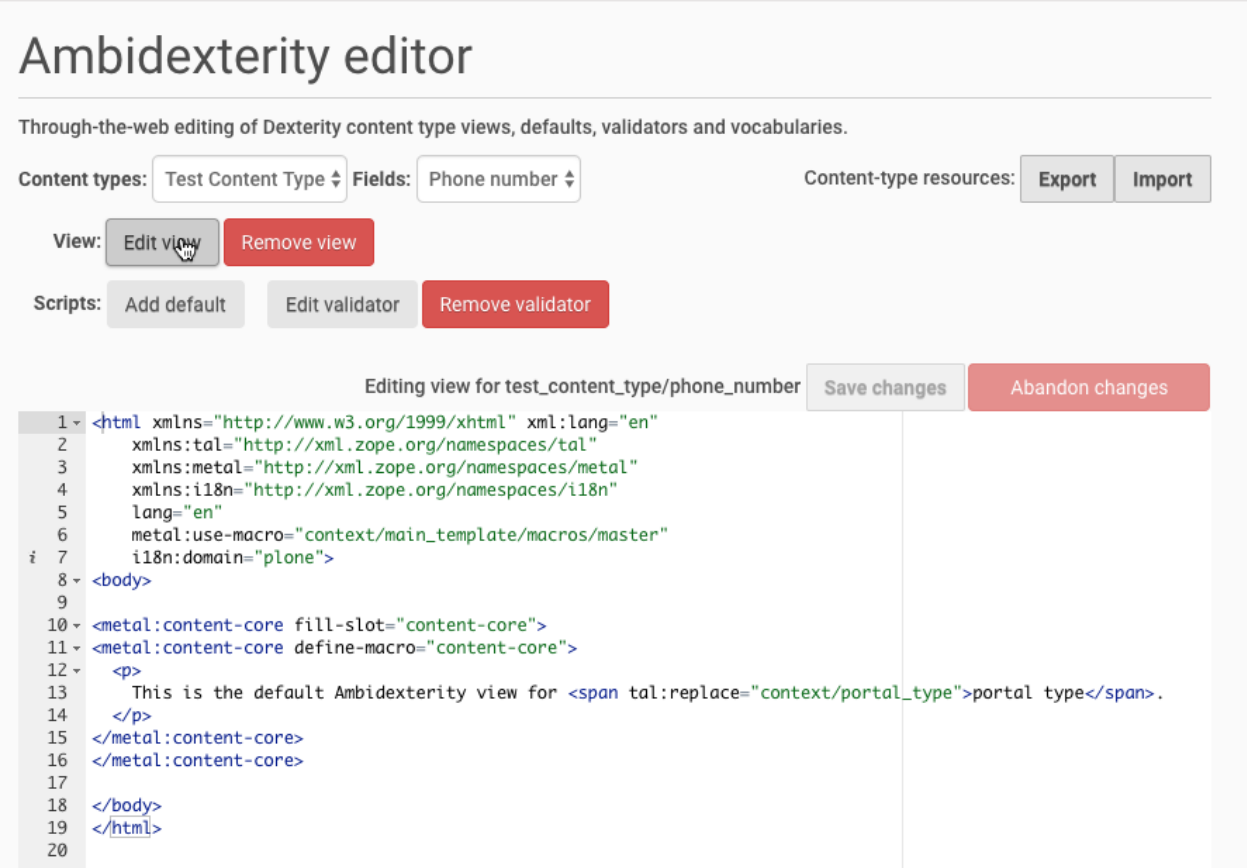

Fig. 9: Editing a view template.

```
<dl>
  <dt>Phone number</dt><dd tal:content="context/phone_number">value</dd>
  <dt>Start date</dt><dd tal:content="context/start_date">value</dd>
</dl>
```

This inserts the value of our two custom fields. Save it away and return to your already added test content item. Now it should look like:

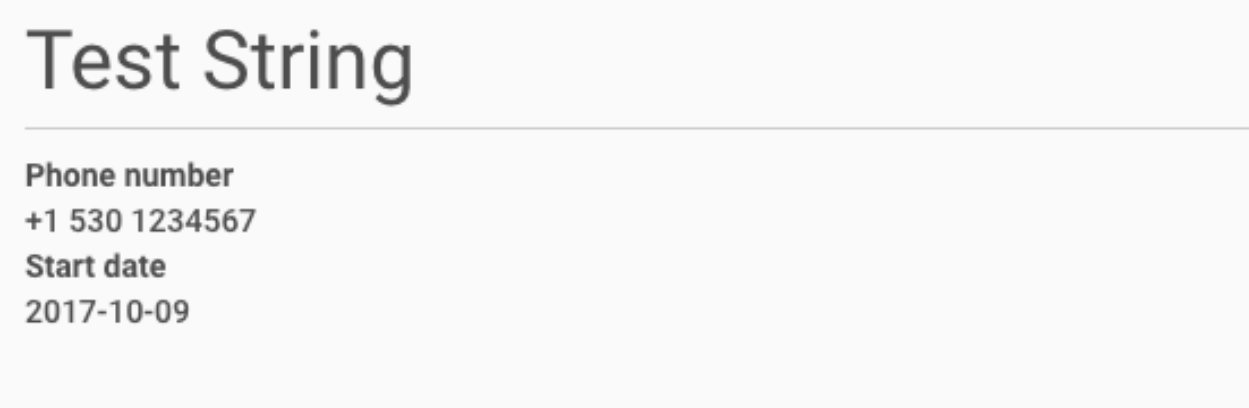

Fig. 10: Our update view.

We've basically recreated the default Dexterity view of the content item. It's up to you to make it fancy.

CHAPTER 2

Export/Import

Ambidexterity includes limited facilities for exporting and importing your content-type views and scripts. The goal is to allow you to elaborate a content type's functionality TTW, then copy those elaborations to a matching content type on another Plone site.

2.1 Background

The views and scripts you add via the Ambidexterity editor have two parts that reside in different parts of the object database of a Plone site. You may examine each via the Zope Management Interface.

The scripts and views themselves reside in `portal_resources` in a folder named `ambidexterity` that has subfolder for each content type and within each content-type folder for fields.

Ambidexterity also changes a content type's Type Profile. You may see those changes by examining your TTW Dexterity types in `portal_types`. You may also see the script-enabling code by viewing the XML for a content type in the Dexterity field editor.

2.2 Exporting a Type Profile

This capacity is built into Dexterity. Visit the Dexterity control panel, check the box to the left of a content type listing, and press the `Export Type Profiles` button. This will generate a zip archive; unpack it if you want to examing it but make sure to keep around the zip file.

2.3 Exporting Ambidexterity resources

Visit the Ambidexterity control panel and select a content type. Press the `Export` button and a zip archive will be downloaded. As with the *Type Profile* zip, you may unpack it to examine it, but keep the zip file.

2.4 Import strategy

I suggest importing an Ambidexterity elaborated content type in four steps:

1. Activate Ambidexterity on the target site;
2. Visit the Dexterity control panel and import your type profile;
3. Visit the Ambidexterity control panel and auto-synchronize when a problem is discovered;
4. In the Ambidexterity editor, choose your imported content type and press the `Import` button to import your Ambidexterity resources.

Step 3 is required by the fact that you have imported a content type that uses Ambidexterity views, classes and functions, but does not yet have any portal resources to match. The auto-synchronization will remove those references from the portal type. Between step 2 and step 3, the content type is broken. Step 3 fixes it, but leaves it with no Ambidexterity support.

Step 4 will automatically re-synchronize the imported Ambidexterity resources with the portal-type information. Everything should work again.

2.5 Caution, caution, caution

Using Ambidexterity should be a quick (and a bit dirty) solution to ad-hoc problems. So, why would you want to transfer Ambidexterity resources from one Plone site to another?

If you have used Dexterity and Ambidexterity to develop a solution you wish to use on multiple Plone sites, you should strongly consider transferring your content-type definition to a Python add-on package. Add-on packages can be version-controlled and tracked; they allow for sophisticated debugging and lack the limitations of RestrictedPython.

CHAPTER 3

About RestrictedPython

All of our scripts are a Python with some special limitations defined by RestrictedPython.

RestrictedPython is meant to provide a safety net for programmers that are not familiar with the Plone/Zope security model. It limits built-in classes, modules and functions. It also controls object-database access, limiting access to items that are available to the current user. The *current user* is not you; if you're using the Ambidexterity editor, you have great powers (and great responsibility). Rather, the *current user* will be the contributor adding or editing the content item.

RestrictedPython is what you're using when you add *Scripts (Python)* in the Zope Management Interface. It's also used by some add-on packages like *PloneFormGen* which allow limited scripting.

3.1 Import restrictions

Python's standard library is rich. The packages installed with Zope and Plone add much, much more. You **really, truly** don't want access to most of that functionality when you're scripting something like a validator, dynamic default or vocabulary. Much of that extra functionality can't be used without some knowledge of the Zope/Plone security model and of the way web requests are handled.

A quick example: you might want to use `urllib` or `urllib2` to use an external resource via http/s to get information for a script. If you do, the network I/O for that request will block execution of the thread processing your request. Request-processing threads are precious on a Plone server; block a few of them and your site is at-least temporarily down.

Want to read something on the file system? Are you prepared to do all the checking to make sure it can't access other information available to the Plone server? Like the salt-hashed passwords of your users?

And, do you trust every possible person that can script on your site to not make those mistakes? Or leave a vulnerability that might allow someone else to edit a script?

We can't vet the whole Python or Zope/Plone libraries for safety. So, we mark a few modules as safe-for-importing in RestrictedPython. The rest are unavailable.

RestrictedPython marks the following modules as safe-for-importing:

- math
- random
- string

Ambidexterity adds two more because they're so obviously useful for defaults and validation:

- datetime
- re

3.1.1 Adding to the safe list

You may add other modules to the safe-for-importing list via add-on package. See collective.localfunctions for an example.

Be cautious. Your changes will affect all functions using RestrictedPython on the Zope instance where you add your modules and types.

3.2 Permissions

Ambidexterity scripts allow you access to a `context` variable that represents the current object. When an item is being added, the context is the folder. When it's being edited, it's the object itself.

With the right permissions, a knowledgeable programmer can work from *context* get access to nearly everything in your object database or request. RestrictedPython controls access to object properties by checking user permissions. In this case, the *user* is the one adding or editing the content object. If a protected attribute or object is read without the right permissions, an *Unauthorized* error is raised, resulting a an HTTP `Forbidden` response and a redirect to the login form.

When you add Ambidexterity scripts, you're working as a very powerful user. Your content contributors should have much less power. So, you're not done with your Ambidexterity scripts until you log in as a less-powerful user to check for authorization errors.

3.3 Just a safety net

RestrictedPython is a safety net. It is not a replacement for caution. The proper use for a safety net is to exercise all caution to avoid falling and only depend on the safety net as a last resort. You should make sure that you do not extend any role that allows Ambidexterity editing to untrusted or uncautious users. (These are the same roles as those that allow creation of Dexterity content types.)

CHAPTER 4

Nuts and Bolts

The Ambidexterity editor is a UI for a mechanism that provides a custom browser view `@@ambidexterityview` and a Dexterity *defaultFactory* `collective.ambidexterity.default`, a *form:validator* `collective.ambidexterity.validate`, and a vocabulary *source* `collective.ambidexterity.vocabulary`.

Each of these knows how to do two tricks when called:

- It introspects the calling environment to deduce the content type and field type; and
- It uses that knowledge to find a matching script or template in `portal_resources`.

Templates are rendered as usual. Scripts are interpreted in `RestrictedPython` and each has special globals. After script execution, the script's local variables are examined to get results.

Ambidexterity views work for all content types. Scripts work only for through-the-web content types where the supermodel XML version of the schema is a property of the factory type information (FTI).

The general idea is that we use Dexterity XML to specify a schema like:

```
<schema>
  <field name="test_integer_field" type="zope.schema.Int">
    <description/>
    <required>False</required>
    <defaultFactory>collective.ambidexterity.default</defaultFactory>
    <title>Test Integer Field</title>
  </field>
  <field name="test_string_field"
    type="zope.schema.TextLine"
    form:validator="collective.ambidexterity.validate"
  >
    <description/>
    <required>False</required>
    <defaultFactory>collective.ambidexterity.default</defaultFactory>
    <title>Test String Field</title>
  </field>
  <field name="test_choice_field" type="zope.schema.Choice">
    <description/>
```

(continues on next page)

(continued from previous page)

```
    <required>False</required>
    <title>Test Choice Field</title>
    <source>collective.ambidexterity.vocabulary</source>
  </field>
</schema>
```

For the Dexterity type "my_simple_type" and we would get:

```
portal_resources/ambidexterity/my_simple_type/test_integer_field/default.py
portal_resources/ambidexterity/my_simple_type/test_string_field/default.py
portal_resources/ambidexterity/my_simple_type/test_string_field/validate.py
portal_resources/ambidexterity/my_simple_type/test_choice_field/vocabulary.py
```

automatically called as appropriate.

4.1 Defaults

The script is given one value (other than standard builtins): "context" – which is either the creation folder if the item is being added or the item if being edited.

The default value should be assigned to "default" in the script and should be of the type required by the field.

4.2 Vocabularies

The script is given one value (other than standard builtins): "context" – which is either the creation folder if the item is being added or the item if being edited.

The vocabulary should be assigned to "vocabulary" in the script. It should be a list of values or a list of items (value, title).

4.3 Validators

The script is given two values (other than standard builtins):

- "context" – which is either the creation folder if the item is being added or the item if being edited.
- "value" – the field value submitted for validation.

If the validator script determines the value is invalid, it should assign an error message to a variable named "error_message".

If the value is valid, do not do either of the above. The absence of an error message is taken to mean all is OK.

4.4 Views

If a view.pt template file is placed at portal_resources/ambidexterity/content_type/view.pt as a text file, it will be usable at @@ambidexterityview.

You may also set other template files and traverse to them at URLs like @@ambidexterityview/custom_file.js. No matter the extension, they will be handled as page templates.