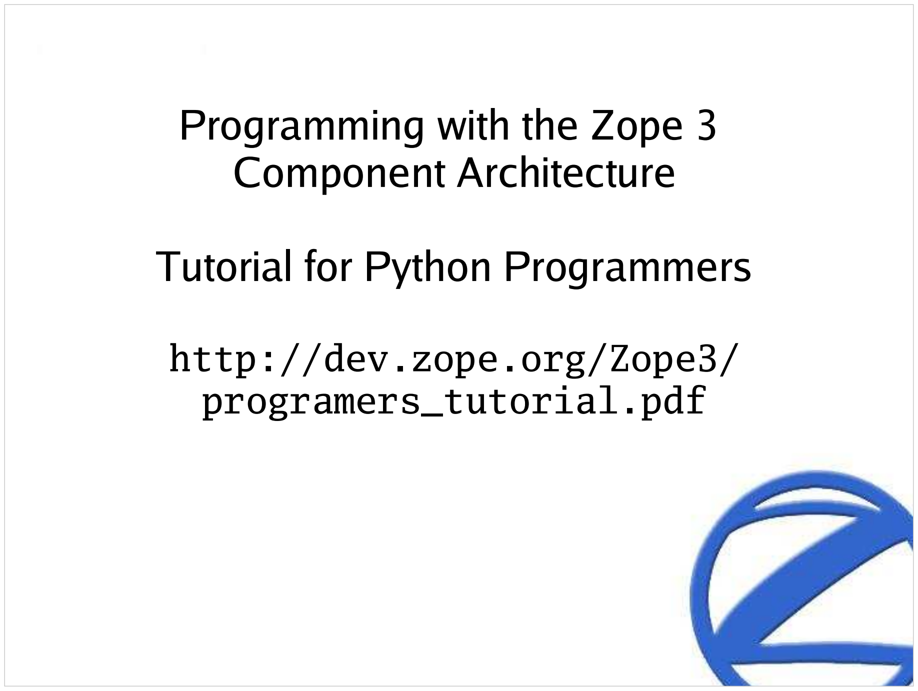


Programming with the Zope 3 Component Architecture

Tutorial for Python Programmers

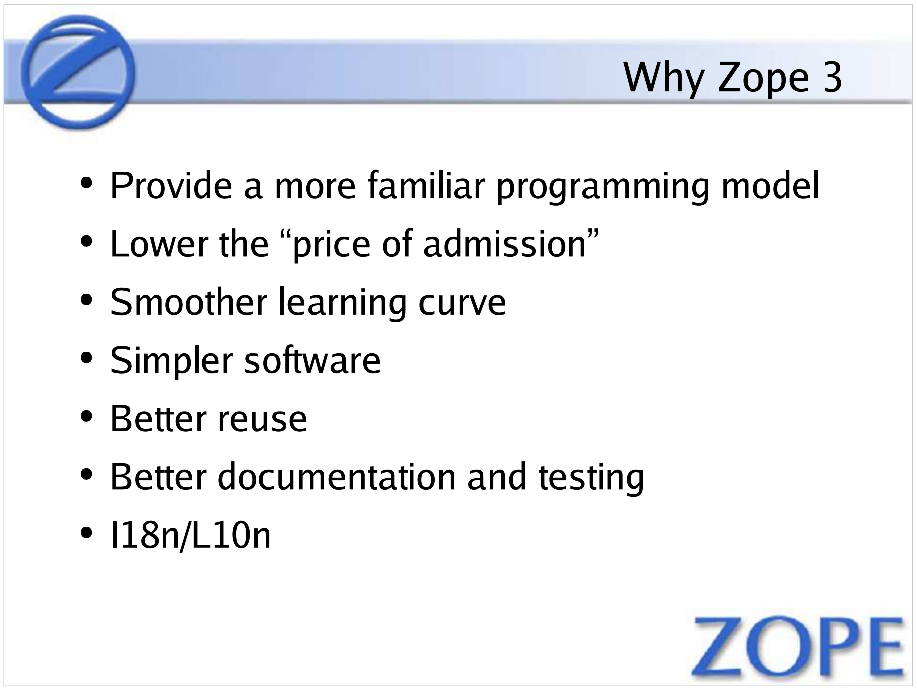
http://dev.zope.org/Zope3/programers_tutorial.pdf

This tutorial provides a short introduction to developing with Zope 3. It provides an example of creating a content objects and associated views, adapters, and utilities.

The skills learned here are applied in most facets of Zope 3 development.

Here are some pointers you might want to refer to when going through this course:

- The Zope 3 web site is http://dev.zope.org/Zope3
- Getting and installing Zope from Subversion: http://dev.zope.org/Zope3/SettingUpAZope3Sandbox
- Coding style: http://dev.zope.org/Zope3/CodingStyle



Why Zope 3

- Provide a more familiar programming model
- Lower the “price of admission”
- Smoother learning curve
- Simpler software
- Better reuse
- Better documentation and testing
- I18n/L10n

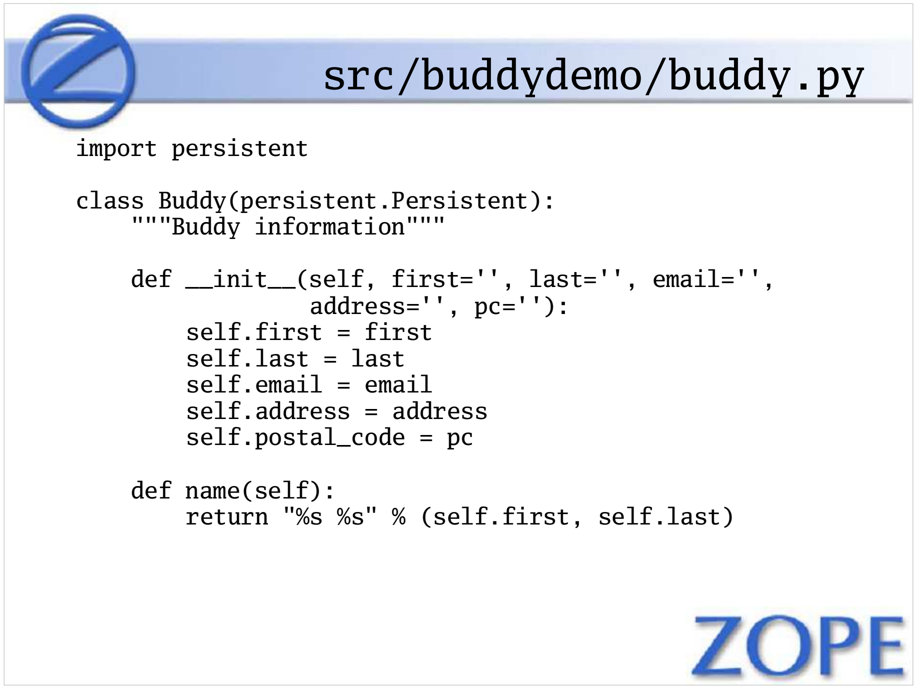
ZOPE

We Zope to be much more approachable to Python programmers.

You should be able to use existing Python objects in Zope with few changes.

We want developers to be able to learn Zope a little bit at a time.

We provide greater support for reuse through components.



src/buddydemo/buddy.py

```
import persistent

class Buddy(persistent.Persistent):
    """Buddy information"""

    def __init__(self, first='', last='', email='',
                 address='', pc=''):
        self.first = first
        self.last = last
        self.email = email
        self.address = address
        self.postal_code = pc

    def name(self):
        return "%s %s" % (self.first, self.last)
```

Let's look at a minimal class that is usable in Zope. As an example, we'll use objects that manage personal information.

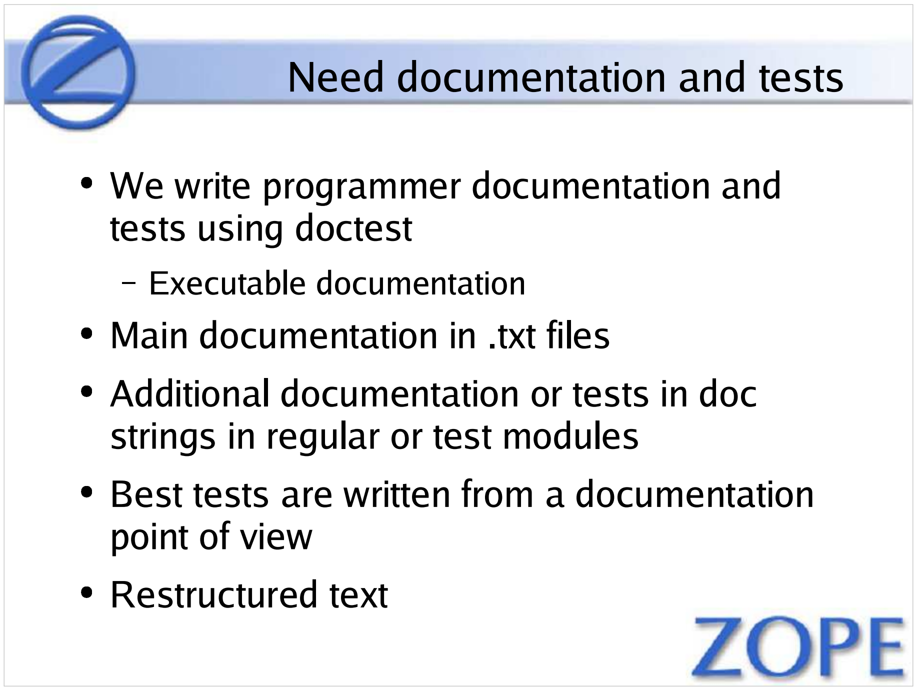
We normally organize our software into packages. We can put our packages anywhere, as long as they are in Zope's Python path.

We'll create a `buddydemo` package in the `src` directory, which is in the Python path. We create an empty `__init__.py` file in `buddydemo`, so that Python will treat buddydemo as a package.

We'll create a `buddy.py` module to hold our class, named `Buddy`.

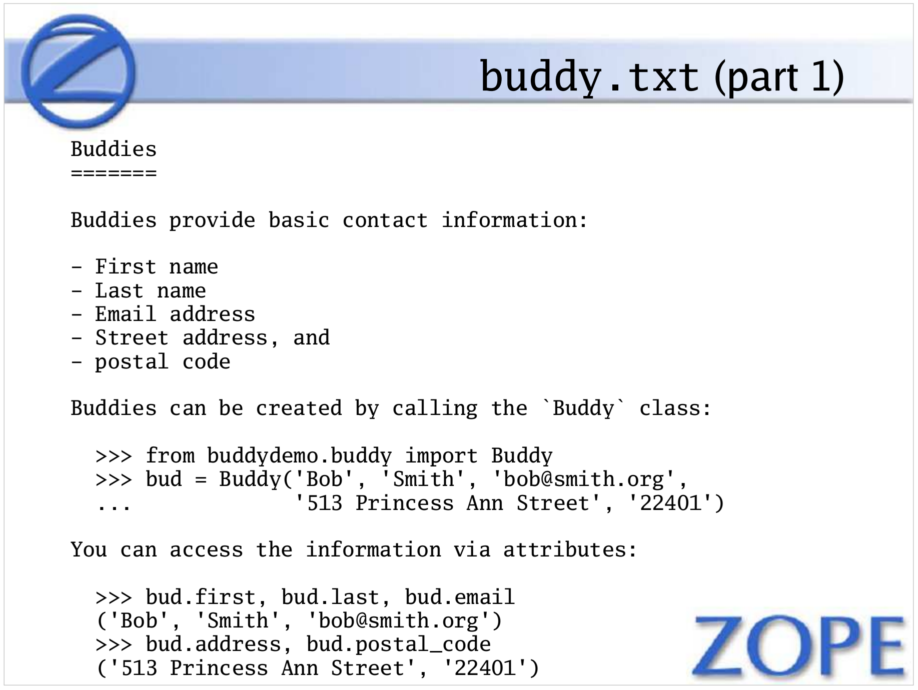
The class is very simple. It stores information in attributes. It provides a single method that combines the first and last name.

There are no Zope-specific mix-in classes. We do subclass `Persistent`. Doing so makes our life easier, because then Zope will manage our data in its object database. We don't have to subclass `Persistent`. If we don't though, we need to manage our data some other way (e.g. in a relational database).



Need documentation and tests

- We write programmer documentation and tests using doctest
  - Executable documentation
- Main documentation in .txt files
- Additional documentation or tests in doc strings in regular or test modules
- Best tests are written from a documentation point of view
- Restructured text



`buddy.txt` (part 1)

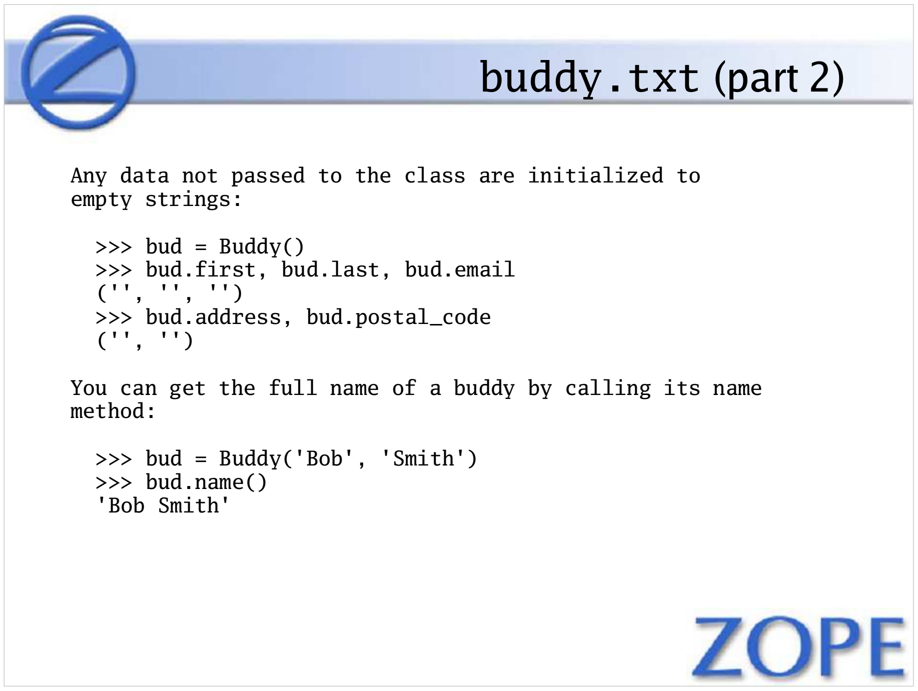
```
Buddies
=======

Buddies provide basic contact information:

- First name
- Last name
- Email address
- Street address, and
- postal code

Buddies can be created by calling the `Buddy` class:

  >>> from buddydemo.buddy import Buddy
  >>> bud = Buddy('Bob', 'Smith', 'bob@smith.org',
  ...             '513 Princess Ann Street', '22401')

You can access the information via attributes:

  >>> bud.first, bud.last, bud.email
  ('Bob', 'Smith', 'bob@smith.org')
  >>> bud.address, bud.postal_code
  ('513 Princess Ann Street', '22401')
```

The Zope project makes heavy use of automated testing. You don't have to write tests to use Zope, but if you want to have high quality software that's easy to change, then you really want to do automated testing.

Zope uses the standard Python `unittest` framework. We also use Python's `doctest` testing facility, which has been integrated with `unittest`. We like writing our tests as `doctest` tests because they help to document our software and make our tests more readable.

Doctest tests are just examples in text files or doc strings that include code that someone might type into an interpreter and what would be printed back. When we test our software, the examples are rerun and the actual output is compared to what's shown in the examples. If the output differs, we get test failures. It's that simple!

Because testing is done by comparing actual and expected output, you need to be a bit careful about what you have in your output:
- The output needs to be the same on each run, so, for example, avoid output with addresses, raw dictionaries and floating-point results.
- Output can exceed 79 characters. This is a problem, because the Zope coding standards call for lines no-longer than 80 characters.
- Blank lines are used by `doctest` to identify the end of output, so the output can't contain blank lines. Use the special marker: <BLANKLINE>
- Watch out for lines with trailing spaces
- Watch out for backslashes. If you have those, use raw doc strings.



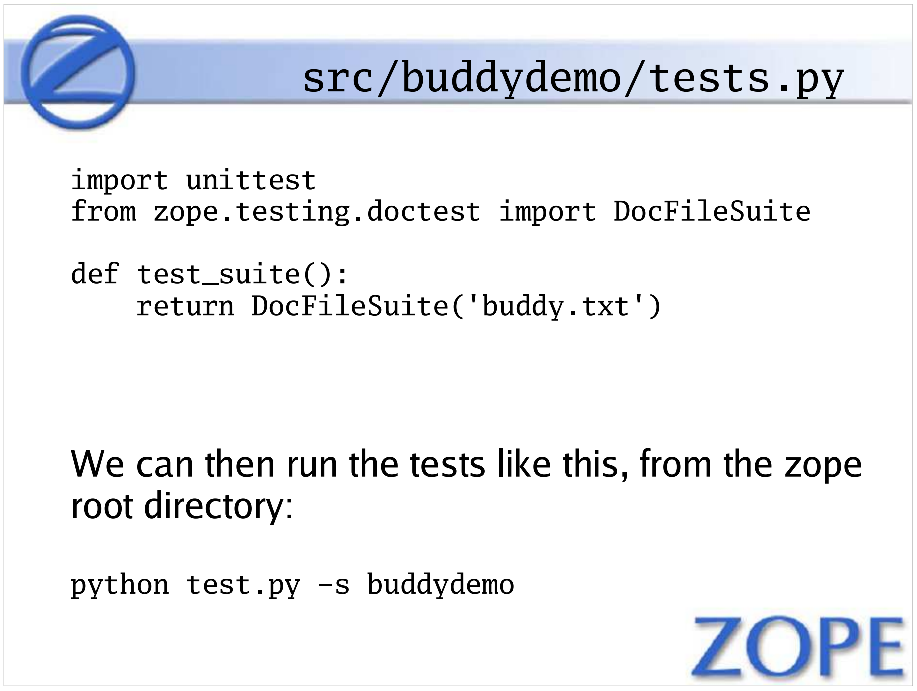
buddy.txt (part 2)

```
Any data not passed to the class are initialized to
empty strings:

  >>> bud = Buddy()
  >>> bud.first, bud.last, bud.email
  ('', '', '')
  >>> bud.address, bud.postal_code
  ('', '')

You can get the full name of a buddy by calling its name
method:

  >>> bud = Buddy('Bob', 'Smith')
  >>> bud.name()
  'Bob Smith'
```

ZOPE

Doctests should tell a story. The main story should go in a .txt file.

Sometimes, you may need to test really odd-ball behaviors that you don't want to discuss in the main documentation. You can use doctrines in the test module for that. We'll show how to do that later.



src/buddydemo/tests.py

```
import unittest
from zope.testing.doctest import DocFileSuite

def test_suite():
    return DocFileSuite('buddy.txt')
```

We can then run the tests like this, from the zope root directory:

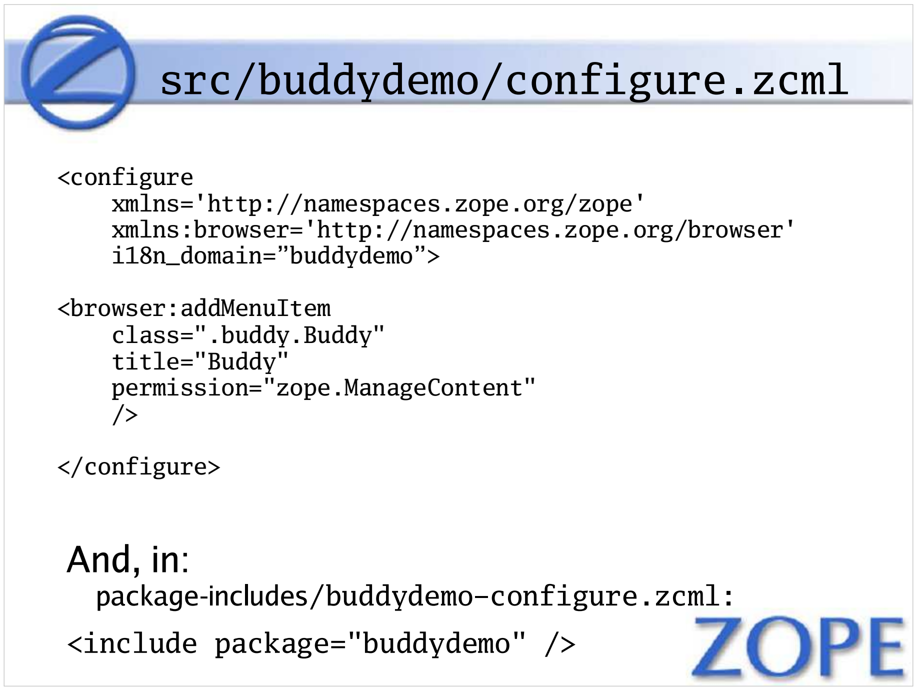
```
python test.py -s buddydemo
```

We need to get our tests run somehow. We'll look at what it takes to get tests run by the Zope test runner. Zope's test runner searches the Zope source tree for modules or packages named "tests". If it finds a module name d `tests`, it will look for a `test_suite` function in module that returns a `unittest` test suite. If it finds a package named `tests`, it will search all of the modules in that package who's names begin with "test".

In this example, we'll create a `tests` module. For now, we just want to run the tests in the buddy module. Our `test_suite` function uses the `DocTestSuite` function to create a test suite from a module. You can pass either a module or a module name.

We run the tests using the Zope test runner, `test.py`. This test runner provides lots of useful features. To find out what they are, run the test runner with a `-h` argument:

```
python test.py -h
```



src/buddydemo/configure.zcml

```
<configure
    xmlns='http://namespaces.zope.org/zope'
    xmlns:browser='http://namespaces.zope.org/browser'
    i18n_domain="buddydemo">

<browser:addMenuItem
    class=".buddy.Buddy"
    title="Buddy"
    permission="zope.ManageContent"
    />

</configure>
```

And, in:

package-includes/buddydemo-configure.zcml:

```
<include package="buddydemo" />
```

To get Zope to use our class, we have to tell Zope about it:

1. We create a configuration file in the package that will accumulate various bits of configuration information.

2. We tell Zope to read our configuration file by including it from a one-line configuration file that we put in the products directory.

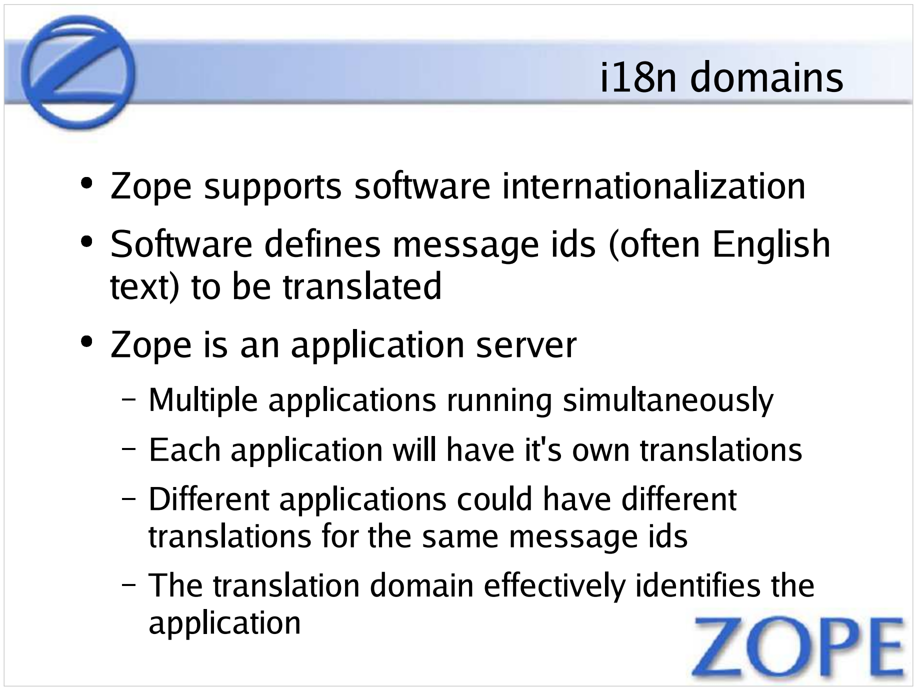
The configuration files are in an XML format called Zope Configuration Markup Language, ZCML. The format is extensible using XML namespaces (and ZCML meta-configuration directives). In our example, we use two namespaces, `zope` (the default), and `browser`. The `browser` namespace is used for configuration directives that specify web-interface information.

In our main configuration file, we use an `addMenuItem` directive to add buddies to the list of things that can be added in the UI.

We use dotted names for two purposes:

1. Dotted names are used to name objects. For example "`.buddy.Buddy`" names the `Buddy` class in the `buddy` module. The leading dot indicates the current package, which is the package containing the ZCML file.

2. Dotted names are used for unique identifiers. When used as identifiers, we base them on packages, but we don't allow them to be shortened. An example of such an identifier is "`zope.ManageContent`".



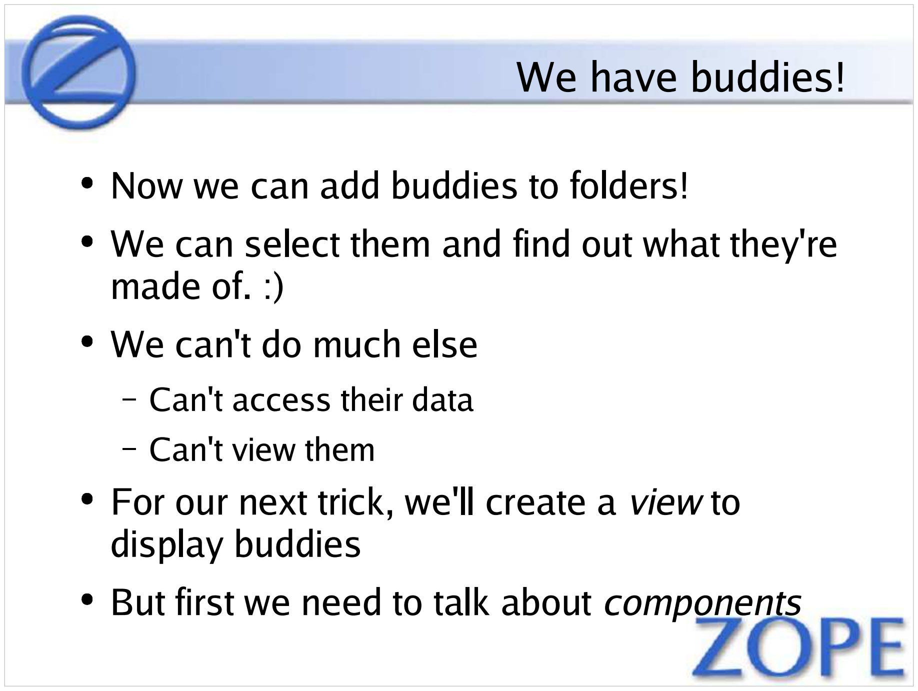
i18n domains

- Zope supports software internationalization
- Software defines message ids (often English text) to be translated
- Zope is an application server
  - Multiple applications running simultaneously
  - Each application will have it's own translations
  - Different applications could have different translations for the same message ids
  - The translation domain effectively identifies the application

Software internationalization (i18n) allows the text defined in our software to be translated. This is distinct from content i18n, which is a different problem, the solution to which will generally depend on particular content-management systems and policies.

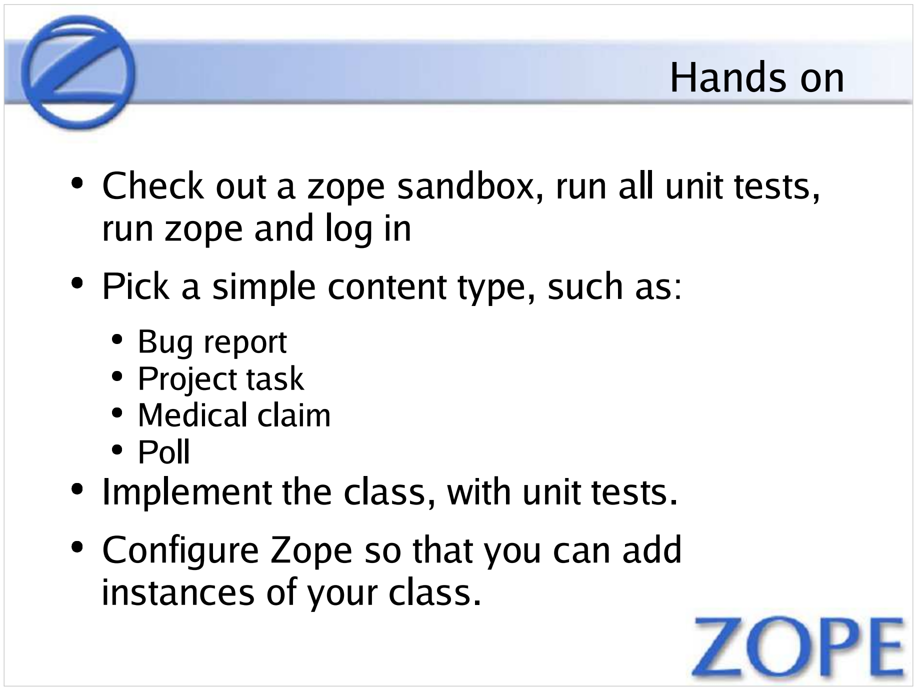
We need to specify a domain whenever we have translatable strings.

The value of the title attribute is a translatable string, so we need to specify a domain in our ZCML file.



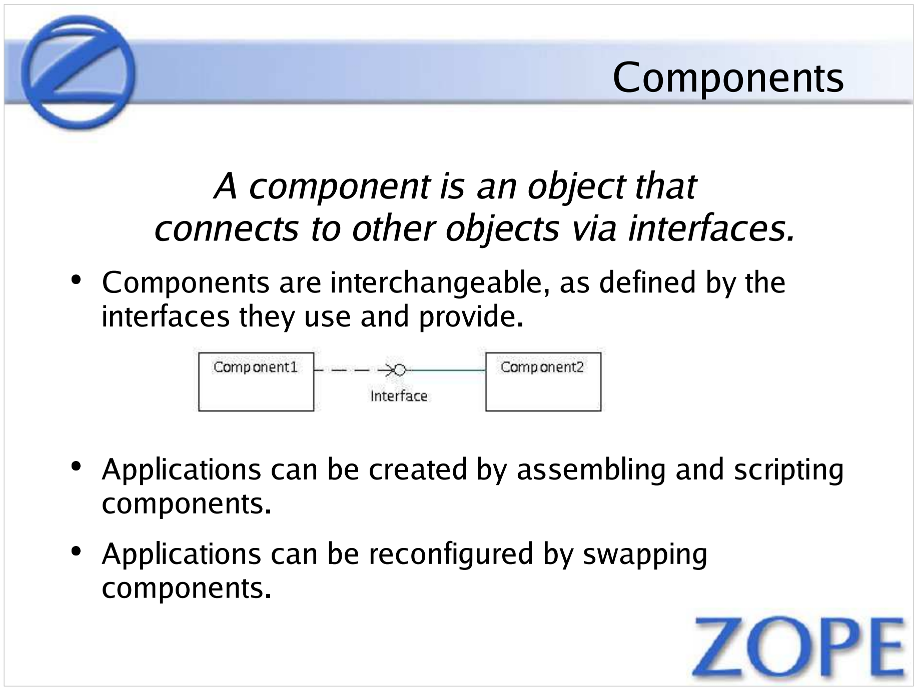
We have buddies!

- Now we can add buddies to folders!
- We can select them and find out what they're made of. :)
- We can't do much else
  - Can't access their data
  - Can't view them
- For our next trick, we'll create a *view* to display buddies
- But first we need to talk about *components*



Hands on

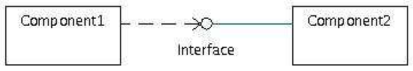
- Check out a zope sandbox, run all unit tests, run zope and log in
- Pick a simple content type, such as:
  - Bug report
  - Project task
  - Medical claim
  - Poll
- Implement the class, with unit tests.
- Configure Zope so that you can add instances of your class.



Components

A component is an object that connects to other objects via interfaces.

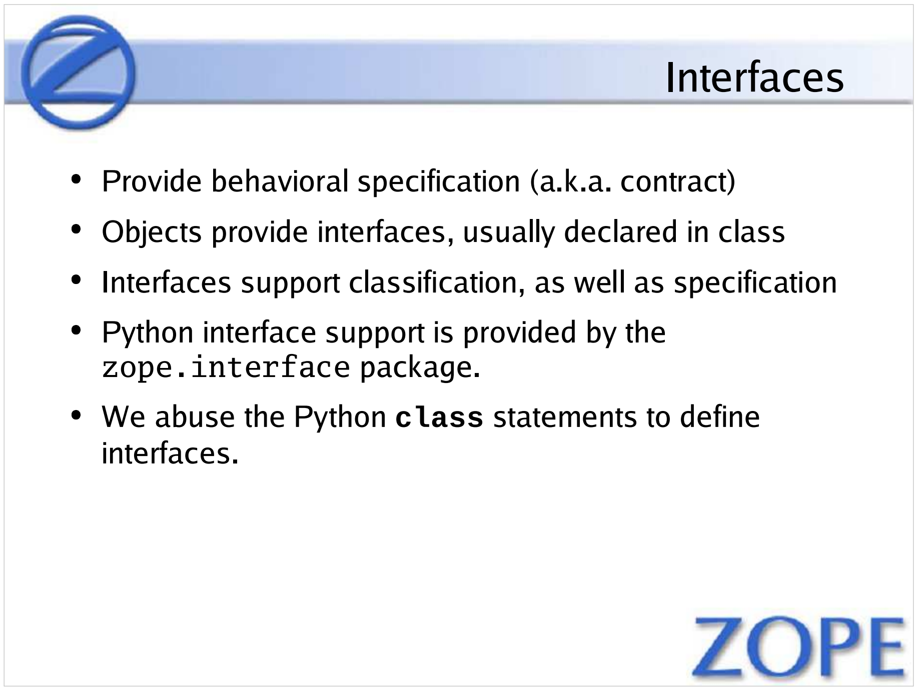
- Components are interchangeable, as defined by the interfaces they use and provide.



- Applications can be created by assembling and scripting components.
- Applications can be reconfigured by swapping components.

ZOPE

What's most important about components is that they get put together with other components to build things. Interfaces provide the basis for connecting things together.

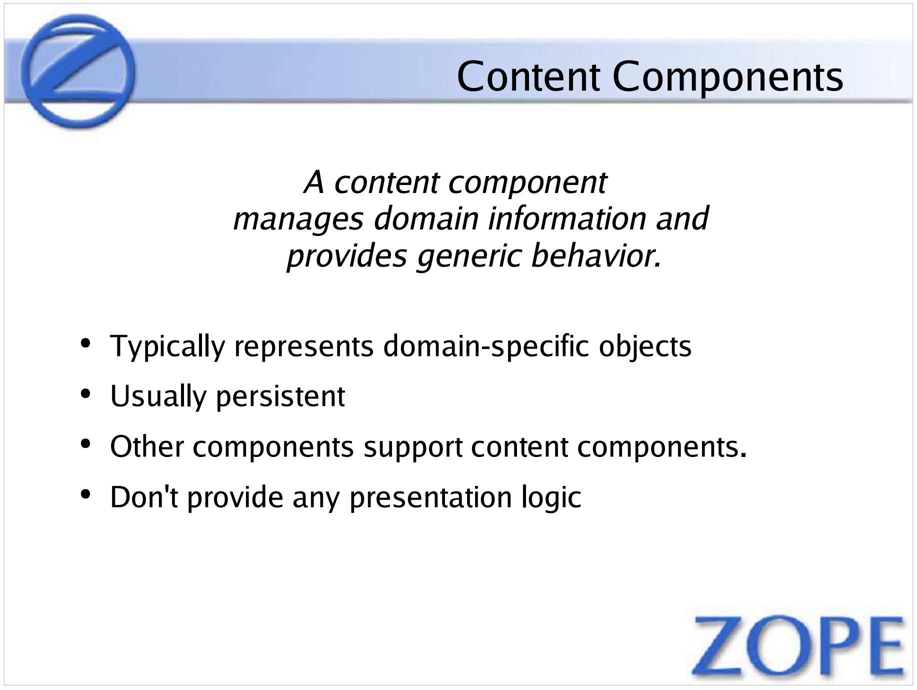


Interfaces

- Provide behavioral specification (a.k.a. contract)
- Objects provide interfaces, usually declared in class
- Interfaces support classification, as well as specification
- Python interface support is provided by the `zope.interface` package.
- We abuse the Python **`class`** statements to define interfaces.

ZOPE

Note that classes are **not** used for strong typing in Zope.

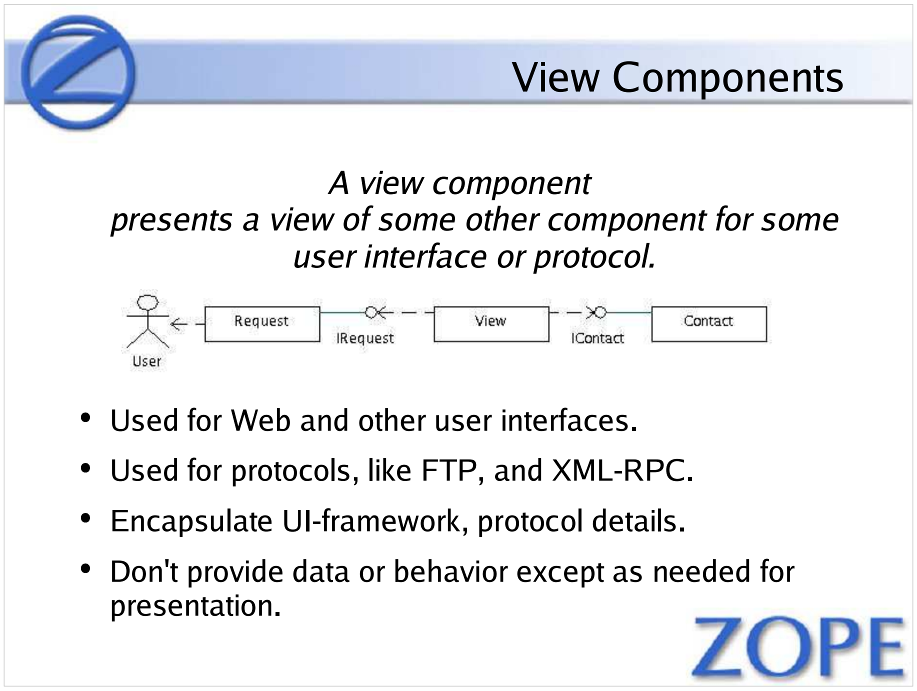

Content Components

A content component manages domain information and provides generic behavior.

- Typically represents domain-specific objects
- Usually persistent
- Other components support content components.
- Don't provide any presentation logic

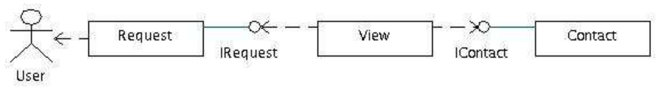
Content components are like:

- Entity beans in J2EE
- Document in Microsoft's Document-View architecture, and like
- Model in the classic Model-View-Controller (MVC) architecture.



View Components

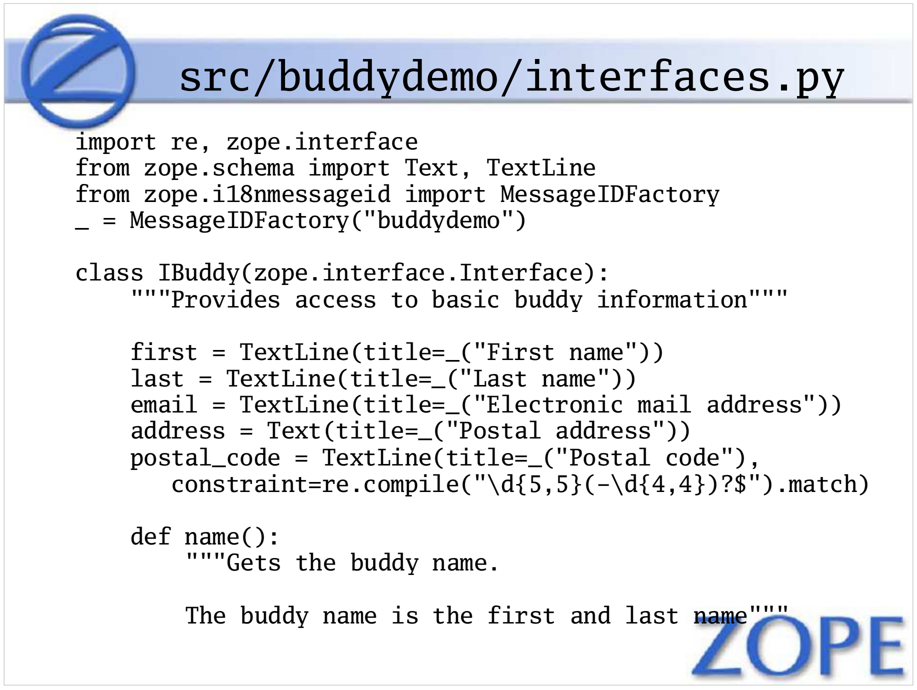
A view component presents a view of some other component for some user interface or protocol.



- Used for Web and other user interfaces.
- Used for protocols, like FTP, and XML-RPC.
- Encapsulate UI-framework, protocol details.
- Don't provide data or behavior except as needed for presentation.

View components connect a user, represented by a request with an object.

Note that view components are a special case of presentation components. Presentation components are responsible for providing user interfaces (or interfaces to other external entities). The other kind of presentation component is resources. Resources are used for thinks like images and style sheets in web interfaces.



src/buddydemo/interfaces.py

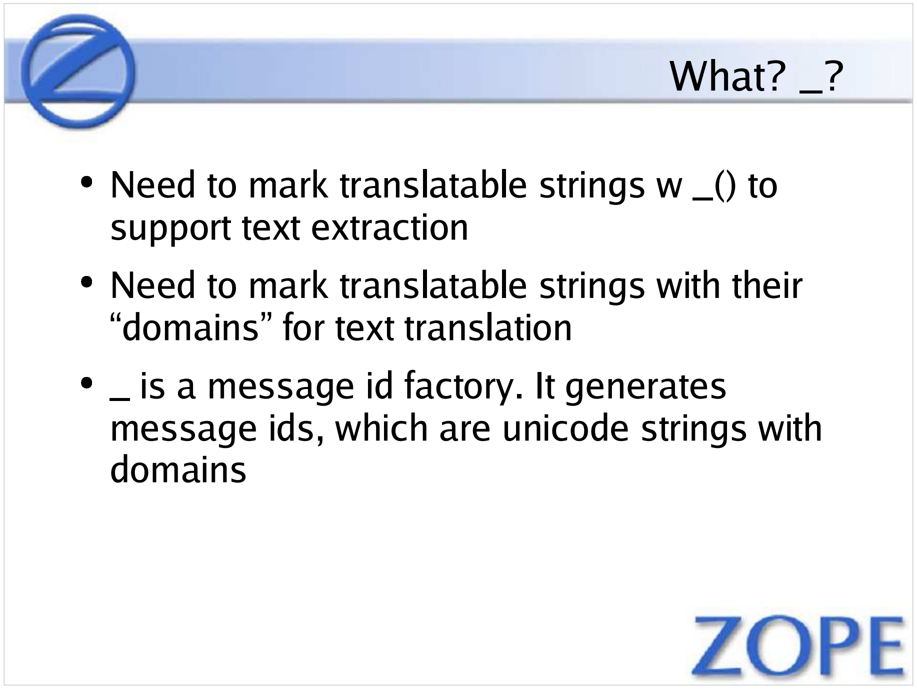
```
import re, zope.interface
from zope.schema import Text, TextLine
from zope.i18nmessageid import MessageIDFactory
_ = MessageIDFactory("buddydemo")

class IBuddy(zope.interface.Interface):
    """Provides access to basic buddy information"""

    first = TextLine(title=_("First name"))
    last = TextLine(title=_("Last name"))
    email = TextLine(title=_("Electronic mail address"))
    address = Text(title=_("Postal address"))
    postal_code = TextLine(title=_("Postal code"),
       constraint=re.compile("\d{5,5}(-\d{4,4})?$").match)

    def name():
        """Gets the buddy name.

        The buddy name is the first and last name"""
```

To make our buddy class a component, we'll create an interface that describes how to use it.

We use the Python `class` statement to create the interface. By subclassing `zope.interface.Interface`, we arrange for an interface **rather** than a class to be created. We could use any and multiple interfaces as base interfaces. Interfaces support multiple inheritance

This interface happens to be a **schema**. A schema is an interface that uses fields to define information attributes, as well as methods. An attribute defined in a schema need not be stored by an implementation. It could be a computed property. Fields define attributes by specifying documentation and **constraints** for the attribute values. In `IBuddy`, we use `Text` and `TextLine` fields. `Text` fields store textual data as unicode. `TextLine` fields are simply `Text` fields without newlines. The `postal_code` field provides an example of using a constraint callable.

When we define methods in interfaces, we don't include “self” arguments. Interfaces specify how objects are **used**. Self parameters aren't passed to methods. They are part of an instance-method's implementation. An interface-specified method might not even be implemented by an instance method (e.g. module functions).



What? _?

- Need to mark translatable strings w _() to support text extraction
- Need to mark translatable strings with their “domains” for text translation
- _ is a message id factory. It generates message ids, which are unicode strings with domains

ZOPE

We rarely ever translate strings directly in Zope. Most strings are translated when they are used to create presentations. We need to somehow record the translation id for a string. We do this by creating message ids. Message ids are unicode strings with extra attributes:

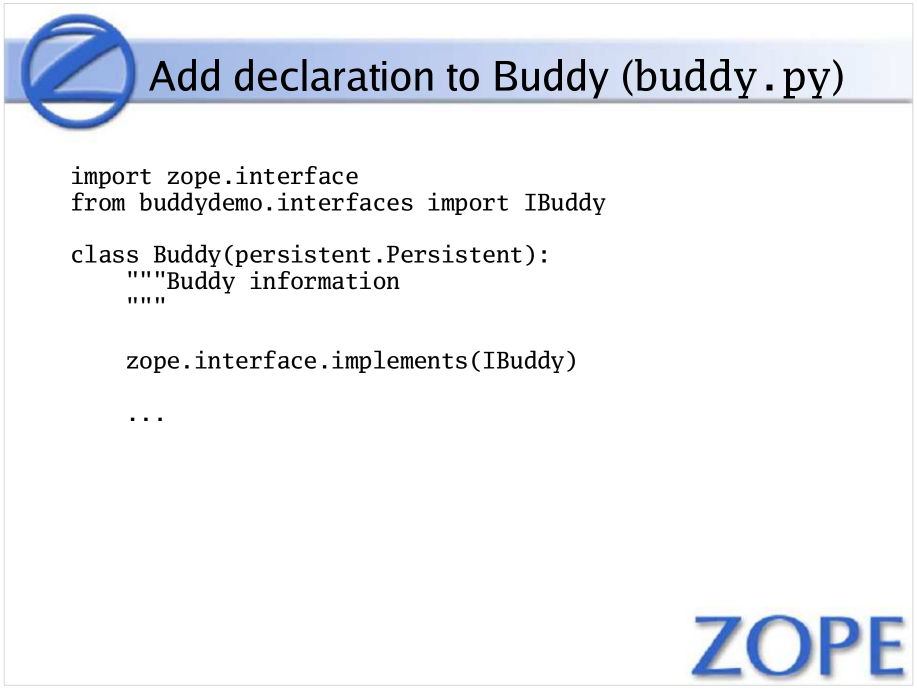
• domain, the message id's translation domain

• default, the text to display if there isn't a translation for the id. If no default is specified, the id is it's own text.

• mapping, a mapping object containing data to be interpolated
Message ids can contain interpolation variables of the form $name, or ${name}. The data to be interpolated can be stored with the string. This would be done in cases where the message ids are being computed dynamically.

We create a message id factory so that we can create message ids by just calling the factory. We use the factory to provide message ids for our schema titles, which will be shown on generated forms.

We name our message id factory “_”. Why? Because standard message-id extraction tools look for translatable strings in Python source by looking for strings passed to “_”.

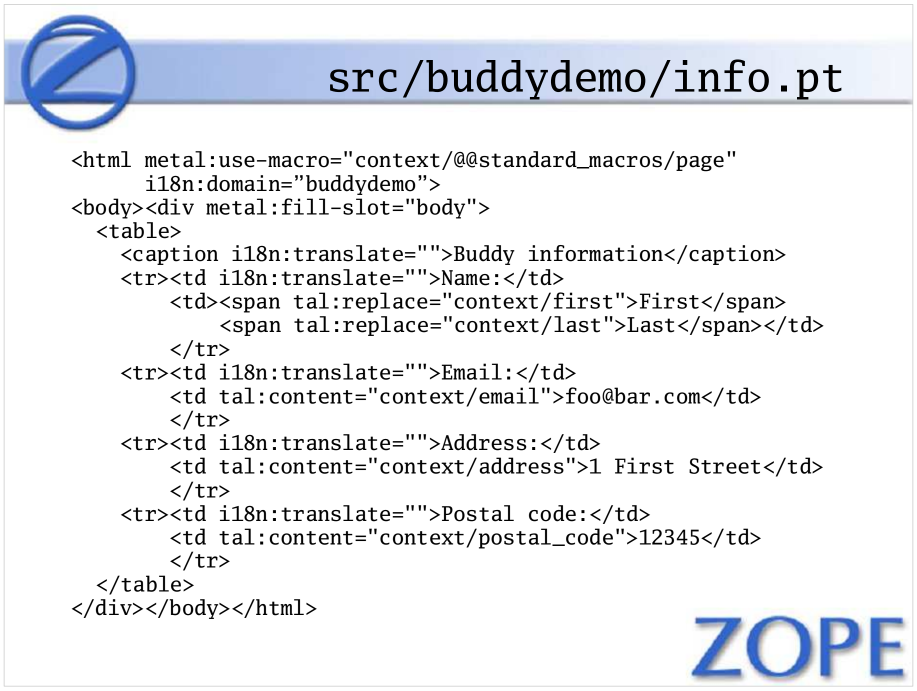


Add declaration to Buddy (buddy.py)

```
import zope.interface
from buddydemo.interfaces import IBuddy

class Buddy(persistent.Persistent):
    """Buddy information
    """

    zope.interface.implements(IBuddy)

    ...
```

We add a declaration to our class saying that the class “implements” IBuddy. When a class implements an interface, that means that the interface can be used to access the classes instances, which “provide” the interface. Note that classes don't implicitly provide the interfaces they implement.

We can also declare that individual objects (e.g. specific instances, classes, modules, etc.) provide interfaces, independent if what their classes implement.



src/buddydemo/info.pt

```
<html metal:use-macro="context/@@standard_macros/page"
      i18n:domain="buddydemo">
<body><div metal:fill-slot="body">
  <table>
    <caption i18n:translate="">Buddy information</caption>
    <tr><td i18n:translate="">Name:</td>
        <td><span tal:replace="context/first">First</span>
            <span tal:replace="context/last">Last</span></td>
        </tr>
    <tr><td i18n:translate="">Email:</td>
        <td tal:content="context/email">foo@bar.com</td>
        </tr>
    <tr><td i18n:translate="">Address:</td>
        <td tal:content="context/address">1 First Street</td>
        </tr>
    <tr><td i18n:translate="">Postal code:</td>
        <td tal:content="context/postal_code">12345</td>
        </tr>
  </table>
</div></body></html>
```

The template will be “bound” to the object being viewed. The view has access to the object being viewed using the `context` variable.

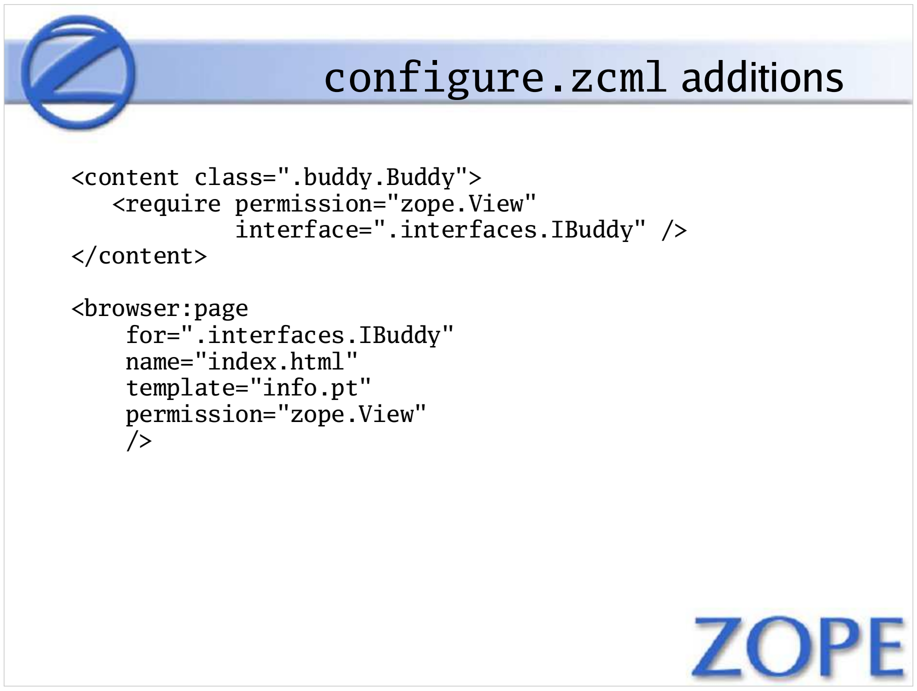
We can access views in path expressions using “@@viewname”. The expression `context/@@standard_macros` looks up a view providing standard macros for the object being views.

Our ZPT contains i18n markup We specify the i18n domain using the `i18n:domain` attribute on the root node. Nodes containing strings to be translated have `i18n:translate` attributes. The `i18n:translate` attribute takes a message id as a value, but the message id can be omitted, in which case the text to be translated is used as the message id. For example, in the name label, the message id is “Name:”. The contents of the node is translated. If the contents included HTML markup, then that would be translated too.

ZPT i18n supports variable interpolation using an `i18n:name` attribute. Here's an example:

```
<p i18n:translate="">Hello <span i18n:name="customer">
                        bob</span>, how are you?</p>
```

In this example, the message id is “Hello $customer, how are you?”.



`configure.zcml` additions

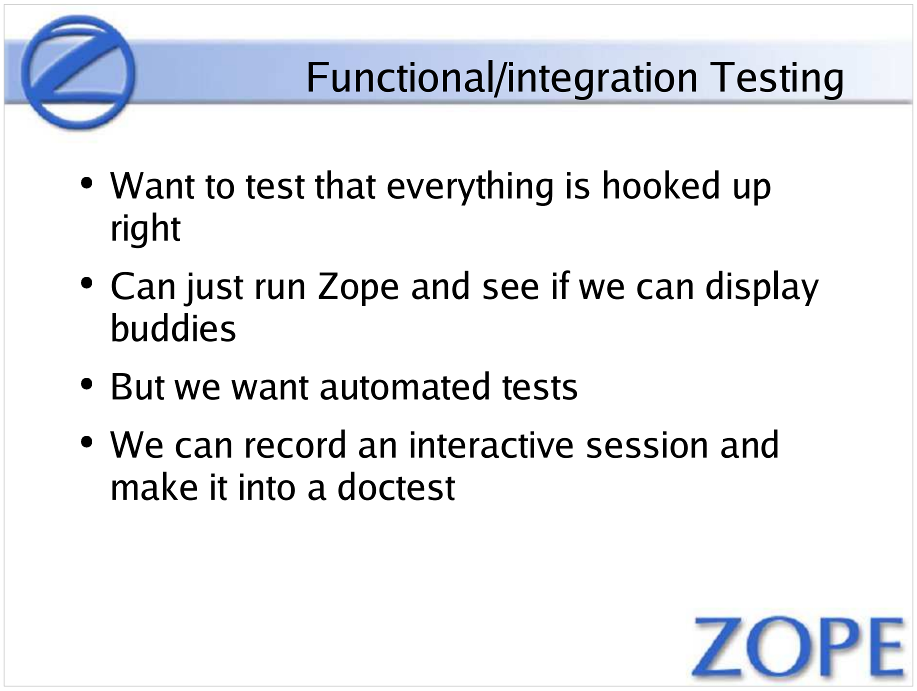
```
<content class=".buddy.Buddy">
   <require permission="zope.View"
            interface=".interfaces.IBuddy" />
</content>

<browser:page
    for=".interfaces.IBuddy"
    name="index.html"
    template="info.pt"
    permission="zope.View"
    />
```

We need to make permission declarations to make it possible for the template to access buddy data. In Zope 3, access protection is provided using security proxies. Because URLs are untrusted code, the results of URL traversal are proxied and the view/template is bound to a security-proxied context. Security proxies forbid access to attributes for which there are no permission declarations. We make permission declarations by saying what permission is required to access one or more names. Names can be specified using interfaces. In this example, we require the `zope.View` permission to access all of the names defined by `IBuddy`.

We define our view using the `browser:page` directive. This actually generates a view class that uses the given template to do it's work. The view will be exposed as a “page” of the content, meaning that we access the view as if the content was a folder and the view was a page in the folder. We specify a name for the page. The name is independent of the template name. Generally, names should have file suffixes to deal with tools that are confused by lack of extensions.

The `for` attribute specifies what kinds of objects the view can be used for. The value of the for attribute is the dotted name of an interface or class.

The `permission` attribute declares what permission is needed to access the view.



Functional/integration Testing

- Want to test that everything is hooked up right
- Can just run Zope and see if we can display buddies
- But we want automated tests
- We can record an interactive session and make it into a doctest

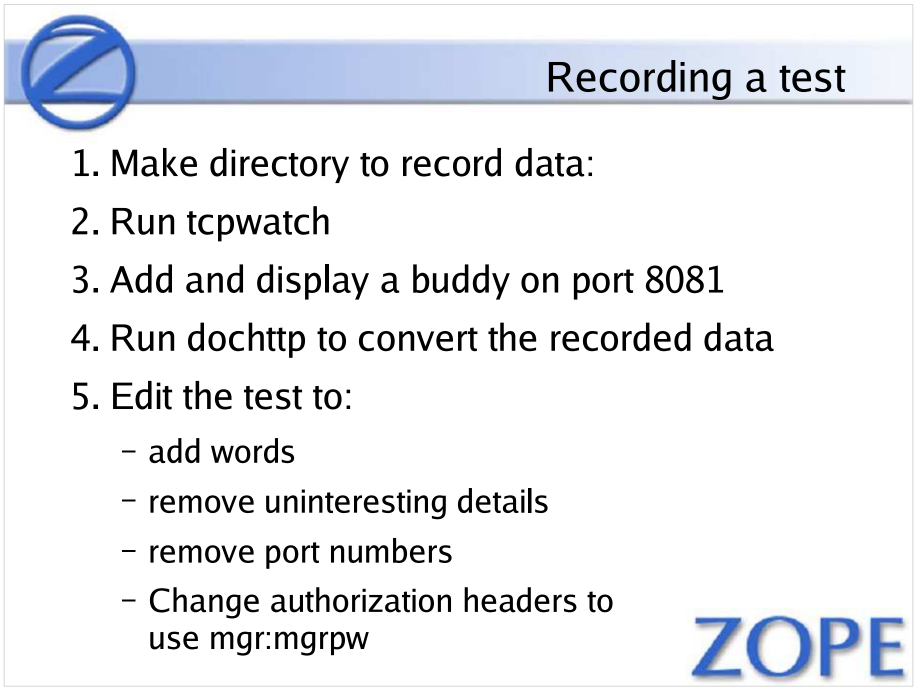
ZOPE

The test we saw before was a “unit” tests. Unit tests test one thing in isolation. We can also write “functional” tests. Functional tests test that the system provides desired functionality at the system boundary -- the user interface. We'll often use functional tests to test that our views, including Python. ZPT and ZCML are working properly together.

We'll use tcpwatch to record our http interactions with our application.

We can get tcpwatch from the Zope cvs at http://cvs.zope.org/Packages/tcpwatch/

We'll also use a script, dochttp, that comes with Zope 3 to convert a recorded http session to a doctest.



Recording a test

1. Make directory to record data:
2. Run tcpwatch
3. Add and display a buddy on port 8081
4. Run dochttp to convert the recorded data
5. Edit the test to:
   - add words
   - remove uninteresting details
   - remove port numbers
   - Change authorization headers to use mgr:mgrpw

To record a session:

```
mkdir record

tcpwatch.py -L8081:localhost:8080 -r record
```

The command above tells tcpwatch to listed on port 8081 and forward to port 8080. (Obviously, you can use different port numbers.)

While tcpwatch is running, start Zope and visit port 8081 (or whatever port you told it to listed on) and perform actions you want to include in the test, such as adding a buddy and displaying it.

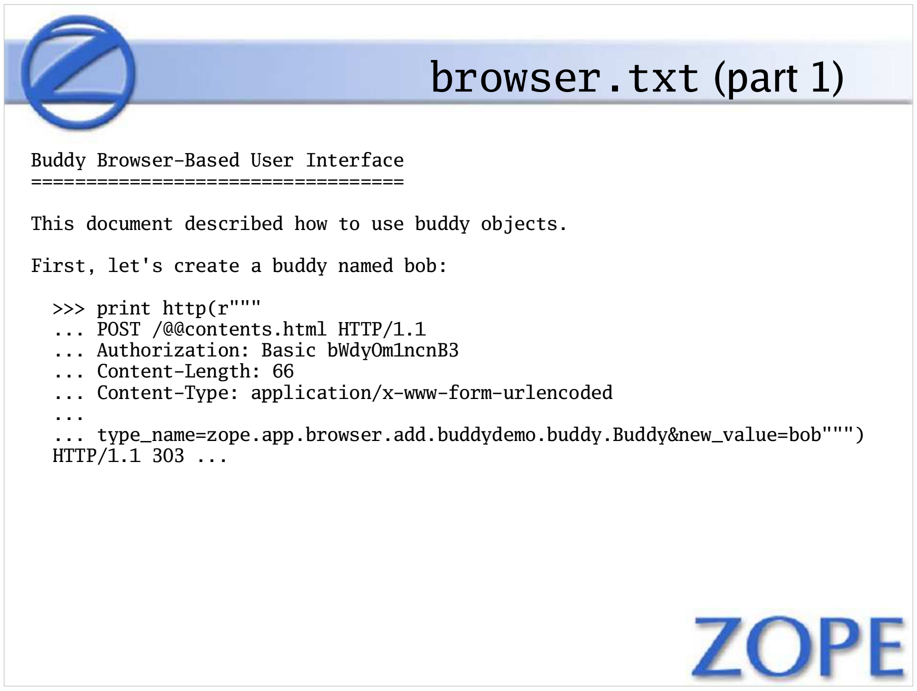
Note that you may also want to disable HTTP connect keep-alive in your browser. Otherwise, you may find the requests and responses appearing out of order.

When you are done recording, exit tcpwatch and run:

```
python src/zope/app/tests/dochttp.py record > browser.txt
```

This will generate a file with lots of data. You will want to find the two requests you care about, the ones that created a buddy and displayed it and remove the other requests from the file.

Edit authorization headers to use “mgr:mgrpw” rather than the base-64 encoded authroization credentials it recorded.



browser.txt (part 1)

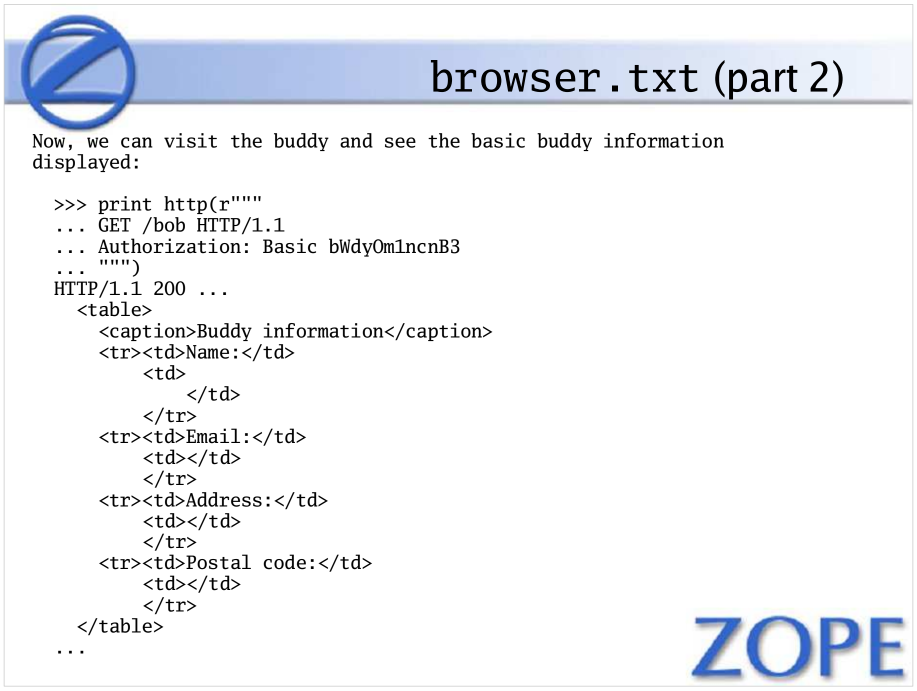
```
Buddy Browser-Based User Interface
==================================

This document described how to use buddy objects.

First, let's create a buddy named bob:

  >>> print http(r"""
  ... POST /@@contents.html HTTP/1.1
  ... Authorization: Basic bWdyOm1ncnB3
  ... Content-Length: 66
  ... Content-Type: application/x-www-form-urlencoded
  ...
  ... type_name=zope.app.browser.add.buddydemo.buddy.Buddy&new_value=bob""")
  HTTP/1.1 303 ...
```

The dochttp script creates a doctest with a series of calls to an “http” function. The function takes an HTTP request message as an input and outputs an an object that, when printed, outputs an HTTP response message. When the test is run, doctest compared the expected and actual response message.

This is the request that adds a buddy. It was recorded by selecting “Buddy” from the add list on the left-hand side of the page. We've removed a Referrer input header that we don't care about and that takes up a lot of space.

We've elided most of the output by adding a “...” after the “303” status code. A 303 status code is used for redirects for HTTP 1.1 clients. The “...” after the 303 is a bit of doctest magic.

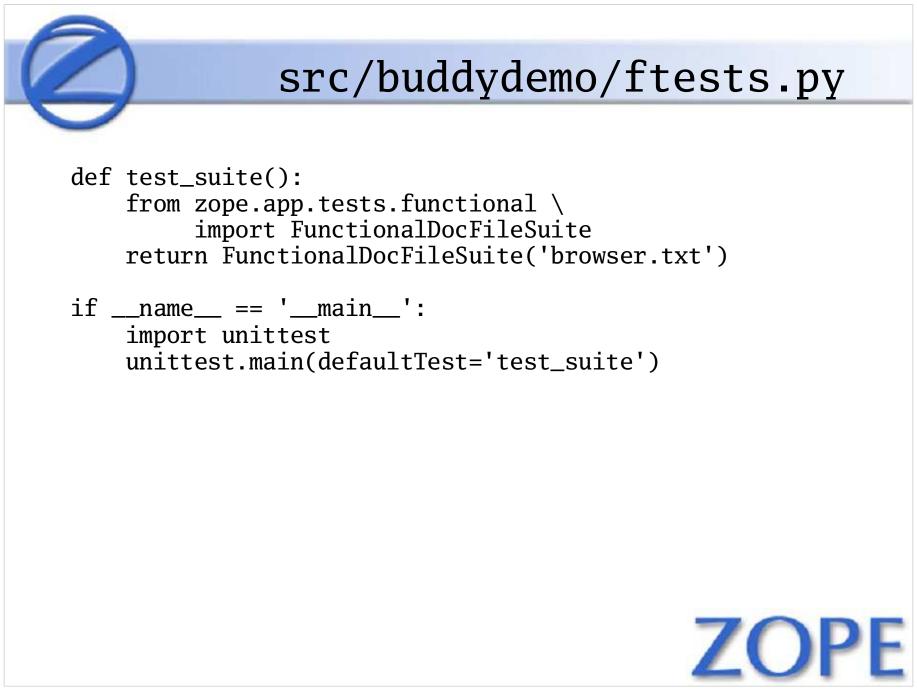
Doctest has a number of options that can be used to effect comparison of actual and expected outputs and to control error-report formatting. Options can be specified in doctest examples themselves and they can be specified when creating tests. Zope's functional-testing doctest support automatically provides the doctest.ELLIPSIS option. This allows “...” in the expected output to match any text. This makes it easy to skip over parts we don't care about.



browser.txt (part 2)

```
Now, we can visit the buddy and see the basic buddy information
displayed:

  >>> print http(r"""
  ... GET /bob HTTP/1.1
  ... Authorization: Basic bWdyOm1ncnB3
  ... """)
  HTTP/1.1 200 ...
    <table>
      <caption>Buddy information</caption>
      <tr><td>Name:</td>
          <td>
              </td>
          </tr>
      <tr><td>Email:</td>
          <td></td>
          </tr>
      <tr><td>Address:</td>
          <td></td>
          </tr>
      <tr><td>Postal code:</td>
          <td></td>
          </tr>
    </table>
  ...
```

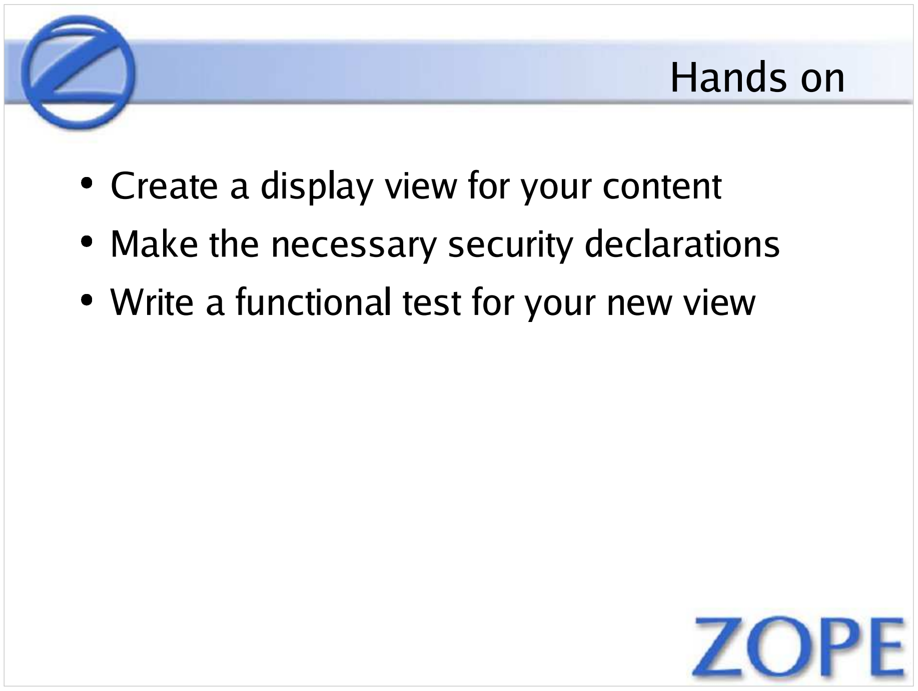
Here, we've elided all of the output except for the table output by the template. Normally, we wouldn't include so much markup, but, at this point, we don't have any interesting data to show.


src/buddydemo/ftests.py

```
def test_suite():
    from zope.app.tests.functional \
         import FunctionalDocFileSuite
    return FunctionalDocFileSuite('browser.txt')

if __name__ == '__main__':
    import unittest
    unittest.main(defaultTest='test_suite')
```

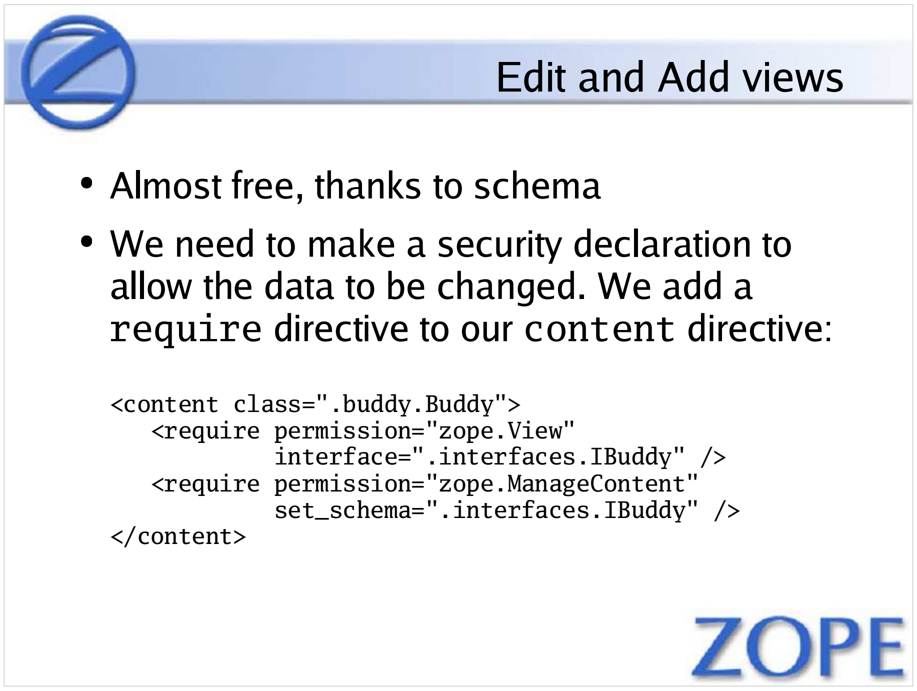
Here, we use FunctionalDocFileSuite, rather that DocFileSuite.

FunctionalDocFileSuite actually sets up a special Zope test server that executes tests.



Hands on

- Create a display view for your content
- Make the necessary security declarations
- Write a functional test for your new view



Edit and Add views

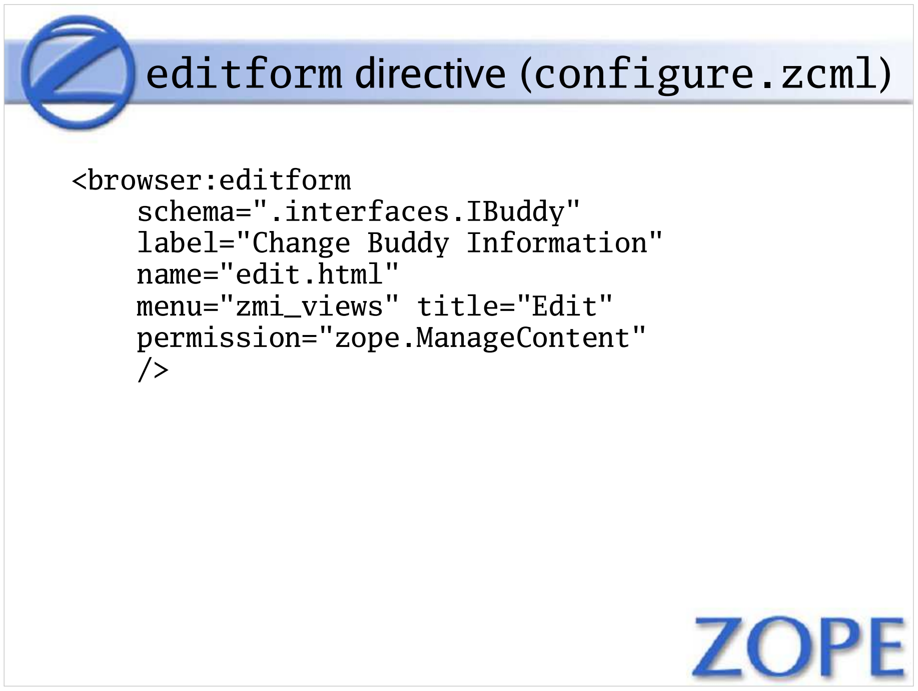
- Almost free, thanks to schema
- We need to make a security declaration to allow the data to be changed. We add a `require` directive to our `content` directive:

```
<content class=".buddy.Buddy">
   <require permission="zope.View"
            interface=".interfaces.IBuddy" />
   <require permission="zope.ManageContent"
            set_schema=".interfaces.IBuddy" />
</content>
```

ZOPE

One of the benefits of schema are that they support automatic form generation. We'll let Zope automatically generate our edit and add forms.

Our edit view will need to be able to modify data by assigning to the attributes defined using fields in the schema. We make permission declarations for assigning to attributes. We can use a `set_attributes` attribute to list attributes to be assigned, or, as we've done here, we can use a `set_schema` attribute. The `set_schema` attribute specified all names that are defined using fields in the schema. In this example, we are not allowing the name attribute to be assigned, because the name attribute is not a field.



editform directive (configure.zcml)

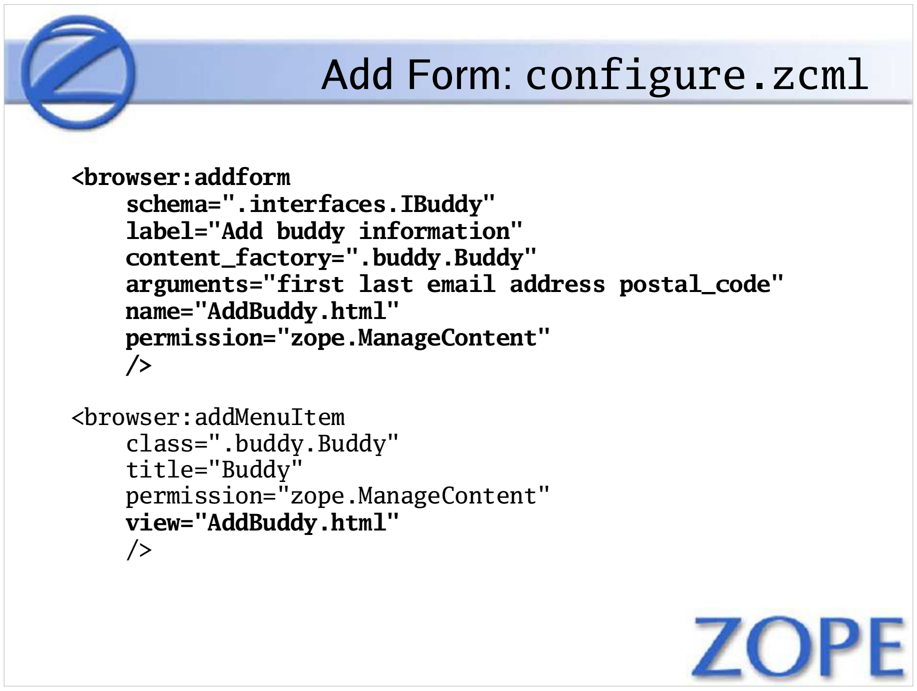
```
<browser:editform
    schema=".interfaces.IBuddy"
    label="Change Buddy Information"
    name="edit.html"
    menu="zmi_views" title="Edit"
    permission="zope.ManageContent"
    />
```

We define our edit view using the editform directive. We specify a schema that specifies the data to be included in the form and the objects the view should be used for. (We could have used a for attribute, to use the view with a different type.) The label attribute allows us to specify a heading for the form. As in the page directive, the name attribute specifies the page name and the permission attribute specifies the permission needed to access the form.

The menu and title attributes are used to add an entry to the zmi_views menu. Zope has a system for defining menus. The zmi_views menu is used for displaying object tabs in the standard management interface. There's a separate menuItem directive. We could have specified the “Edit” menu item separately:

```
<menuItem
  menu="zmi_views" title="Edit"
  for=".interfaces.Buddy"
  action="edit.html"
  permission="zope.ManageContent" />
  />
```

Specifying the menu item in the view definition is a convenient short cut. The menuItem directive provides additional options that are sometimes useful.



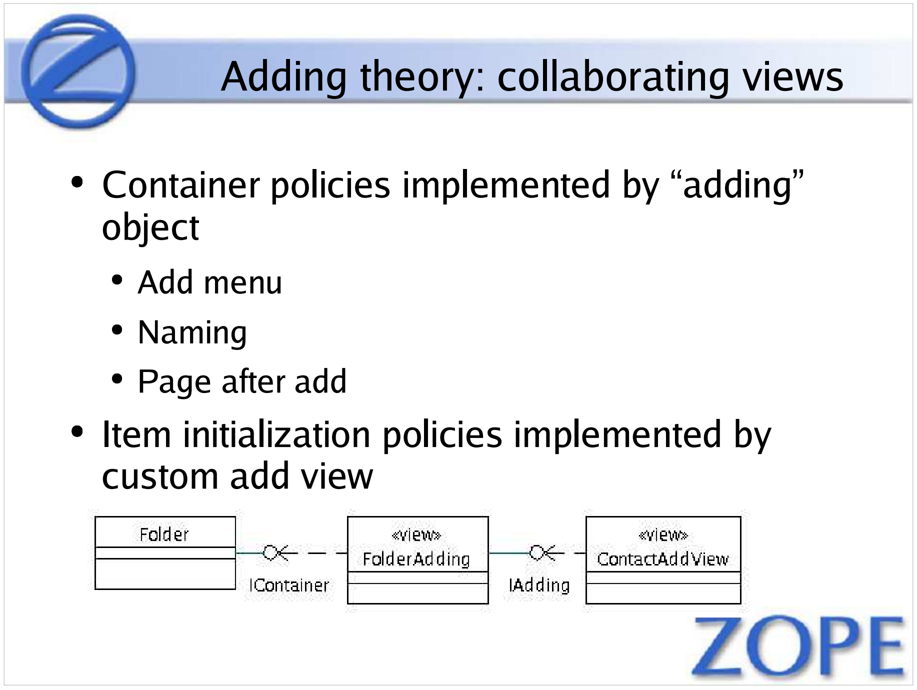
Add Form: configure.zcml

```
<browser:addform
    schema=".interfaces.IBuddy"
    label="Add buddy information"
    content_factory=".buddy.Buddy"
    arguments="first last email address postal_code"
    name="AddBuddy.html"
    permission="zope.ManageContent"
    />

<browser:addMenuItem
    class=".buddy.Buddy"
    title="Buddy"
    permission="zope.ManageContent"
    view="AddBuddy.html"
    />
```

We define an add form using an `addform` directive. The `schema` specifies the data to be collected. The `label` specifies a form heading.

We need to specify a factory for creating the object to be added. We specify this using the `content_factory` attribute. The factory may require arguments. We can specify positional arguments using the `arguments` attribute. The value of the attribute is a list of field names. The corresponding data collected in the form is passed in the given order. We can also specify keyword arguments. In this example, we didn't need to specify arguments. The Buddy class doesn't require arguments to it's constructor. Any fields not specified as arguments will be assigned as attributes after the object is created.

Having defined an add form, we need to modify our `addMenuItem` directive and specify the name of the add view. When someone selects the menu item, the given view will be displayed.



Adding theory: collaborating views

- Container policies implemented by “adding” object
  - Add menu
  - Naming
  - Page after add
- Item initialization policies implemented by custom add view

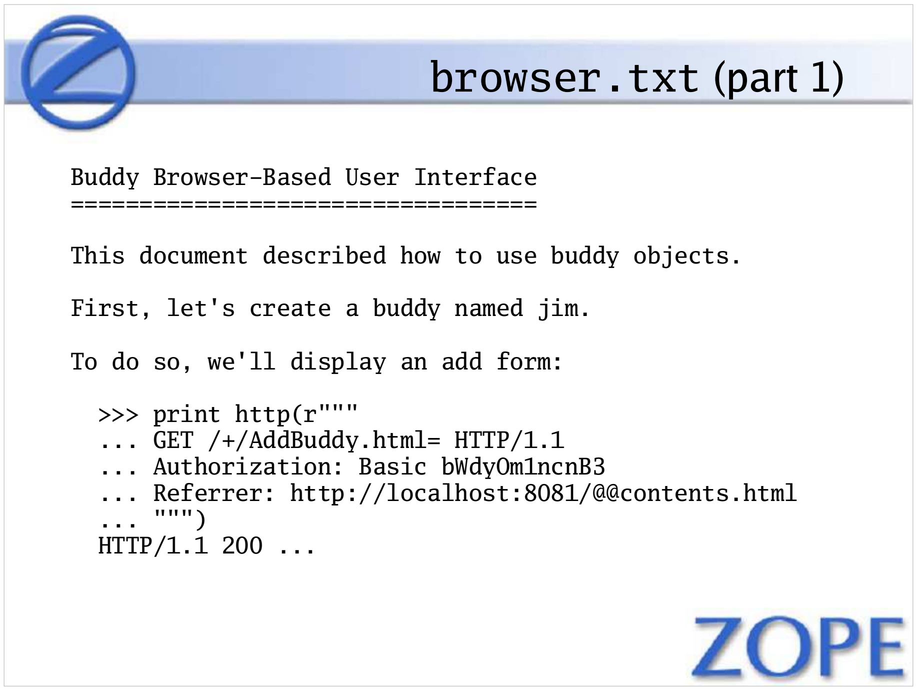
Folder
IContainer
«view»
FolderAdding
IAdding
«view»
ContactAddView
ZOPE

Something to be aware of is that add views are views of special objects called “adding” objects. Adding objects are actually views, so when we add objects, we are using views of views. Why? Each view represents different interests. The adding view represents the interests of the container. The add view represents the interests of the object being added.

When objects are added, there is an entry in the URL for each view. For example, when adding a buddy to a root folder, we'll have a URL like:

```
http://localhost:8080/+/AddBuddy.html=
```

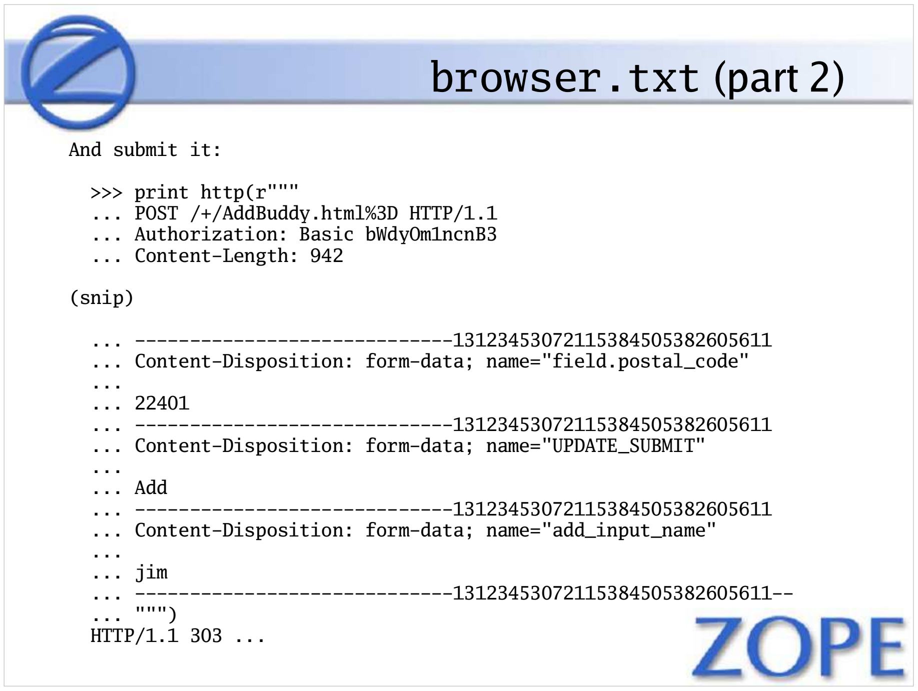
The “+” in the URL is the name of the adding view. The “AddBuddy.html” is the name of the add view.



browser.txt (part 1)

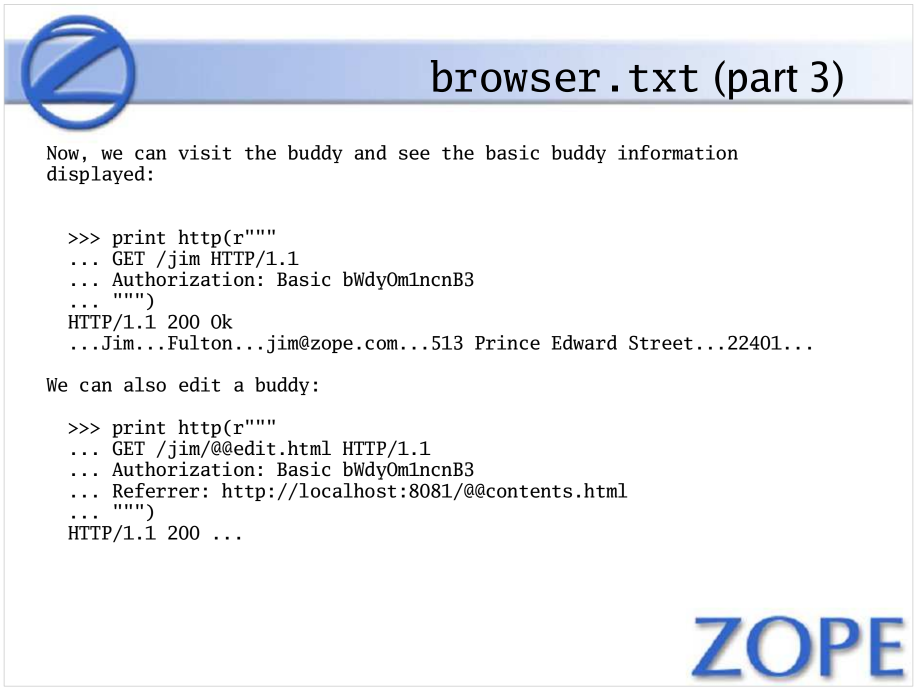
```
Buddy Browser-Based User Interface
==================================

This document described how to use buddy objects.

First, let's create a buddy named jim.

To do so, we'll display an add form:

  >>> print http(r"""
  ... GET /+/AddBuddy.html= HTTP/1.1
  ... Authorization: Basic bWdyOm1ncnB3
  ... Referrer: http://localhost:8081/@@contents.html
  ... """)
  HTTP/1.1 200 ...
```

We have to redo the test, because buddies are added differently. Fortunately, this is pretty easy, as we simply record a session, add some words and delete bits we don't care about.


browser.txt (part 2)

```
And submit it:

  >>> print http(r"""
  ... POST /+/AddBuddy.html%3D HTTP/1.1
  ... Authorization: Basic bWdyOm1ncnB3
  ... Content-Length: 942

(snip)

  ... -----------------------------13123453072115384505382605611
  ... Content-Disposition: form-data; name="field.postal_code"
  ...
  ... 22401
  ... -----------------------------13123453072115384505382605611
  ... Content-Disposition: form-data; name="UPDATE_SUBMIT"
  ...
  ... Add
  ... -----------------------------13123453072115384505382605611
  ... Content-Disposition: form-data; name="add_input_name"
  ...
  ... jim
  ... -----------------------------13123453072115384505382605611--
  ... """)
  HTTP/1.1 303 ...
```

The request didn't fit on a the slide. I've cut out part of the input, indicated by the “(snip)”.



browser.txt (part 3)

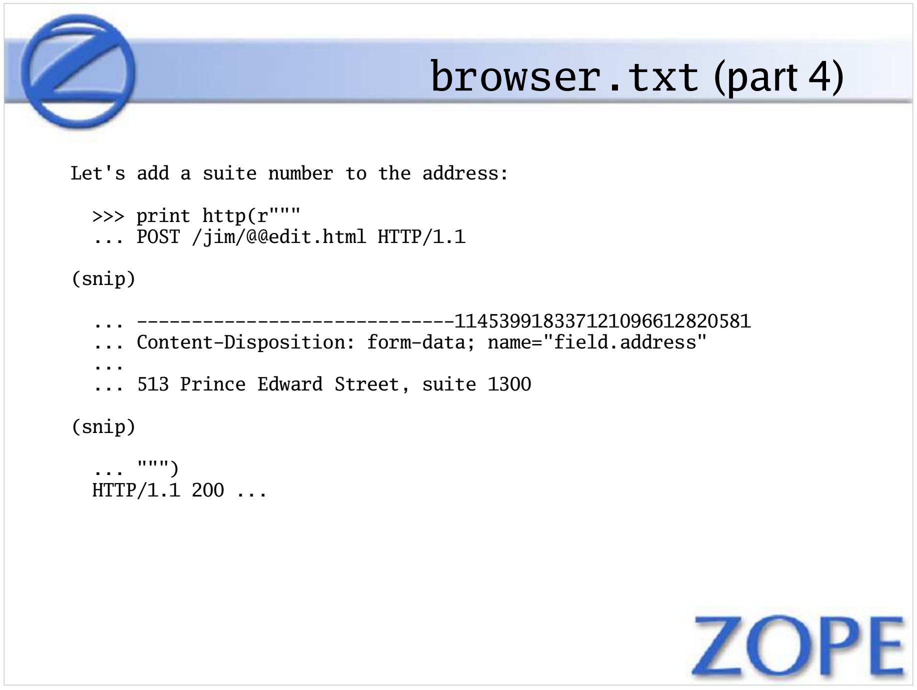
Now, we can visit the buddy and see the basic buddy information displayed:

```
  >>> print http(r"""
  ... GET /jim HTTP/1.1
  ... Authorization: Basic bWdyOm1ncnB3
  ... """)
  HTTP/1.1 200 Ok
  ...Jim...Fulton...jim@zope.com...513 Prince Edward Street...22401...
```

We can also edit a buddy:

```
  >>> print http(r"""
  ... GET /jim/@@edit.html HTTP/1.1
  ... Authorization: Basic bWdyOm1ncnB3
  ... Referrer: http://localhost:8081/@@contents.html
  ... """)
  HTTP/1.1 200 ...
```

Now we have some interesting data, so we show just the data and omit the markup, which is subject to change.

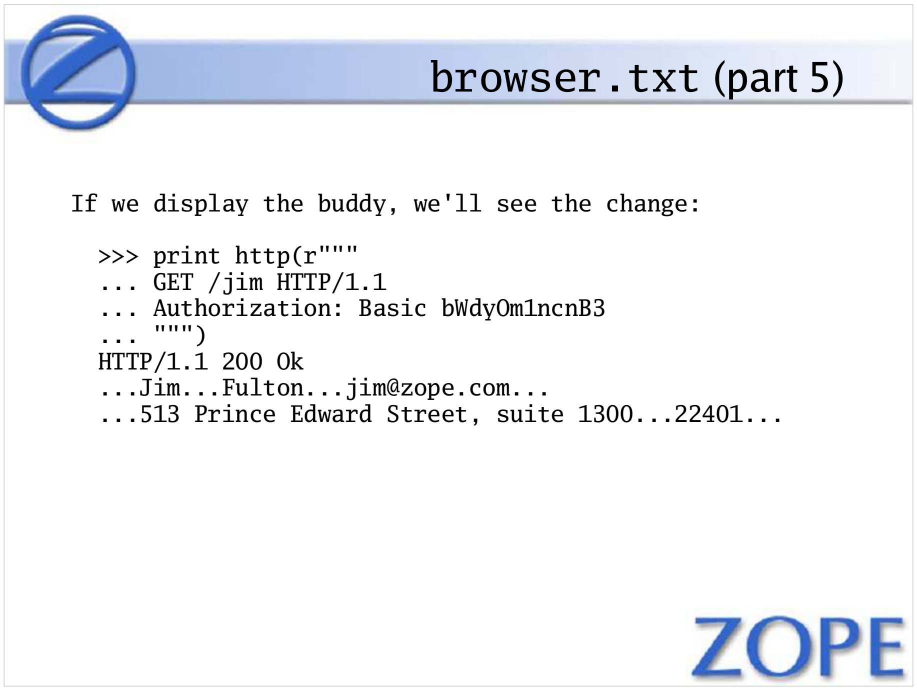


browser.txt (part 4)

```
Let's add a suite number to the address:

  >>> print http(r"""
  ... POST /jim/@@edit.html HTTP/1.1

(snip)

  ... -----------------------------114539918337121096612820581
  ... Content-Disposition: form-data; name="field.address"
  ...
  ... 513 Prince Edward Street, suite 1300

(snip)

  ... """)
  HTTP/1.1 200 ...
```

ZOPE

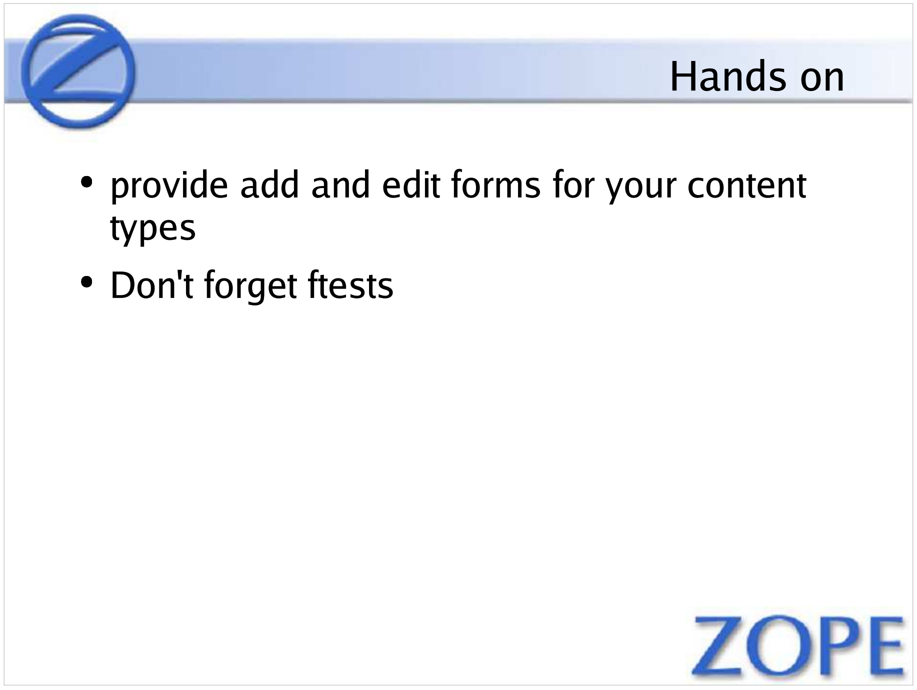
The request didn't fit on a the slide. I've cut out part of the input, indicated by the “(snip)”.



browser.txt (part 5)

If we display the buddy, we'll see the change:

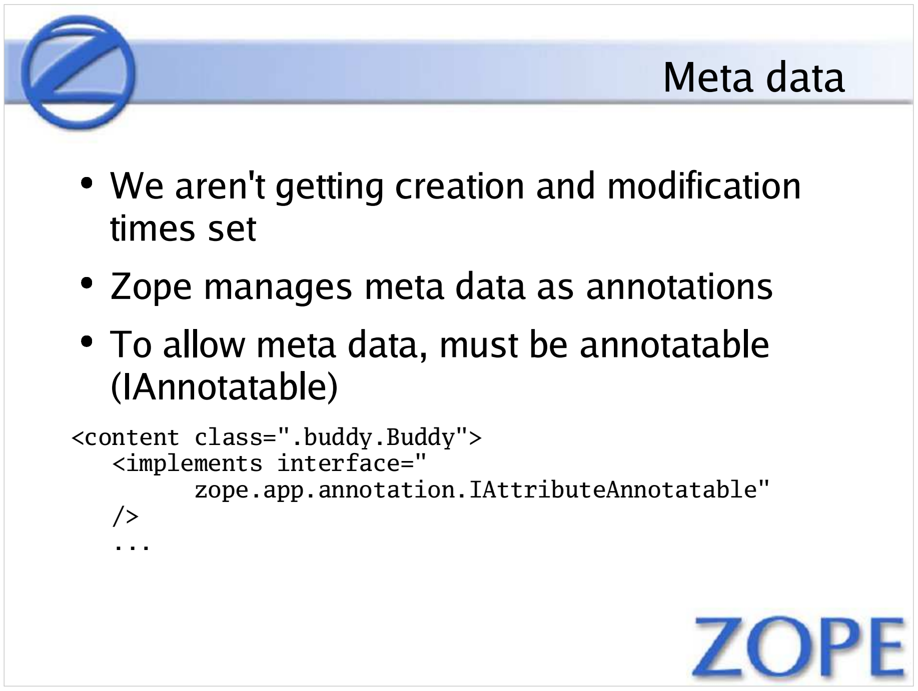
```
  >>> print http(r"""
  ... GET /jim HTTP/1.1
  ... Authorization: Basic bWdyOm1ncnB3
  ... """)
  HTTP/1.1 200 Ok
  ...Jim...Fulton...jim@zope.com...
  ...513 Prince Edward Street, suite 1300...22401...
```



Hands on

- provide add and edit forms for your content types
- Don't forget ftests

ZOPE


Meta data

- We aren't getting creation and modification times set
- Zope manages meta data as annotations
- To allow meta data, must be annotatable (IAnnotatable)

```
<content class=".buddy.Buddy">
   <implements interface="
         zope.app.annotation.IAttributeAnnotatable"
   />
   ...
```

Some definitions:

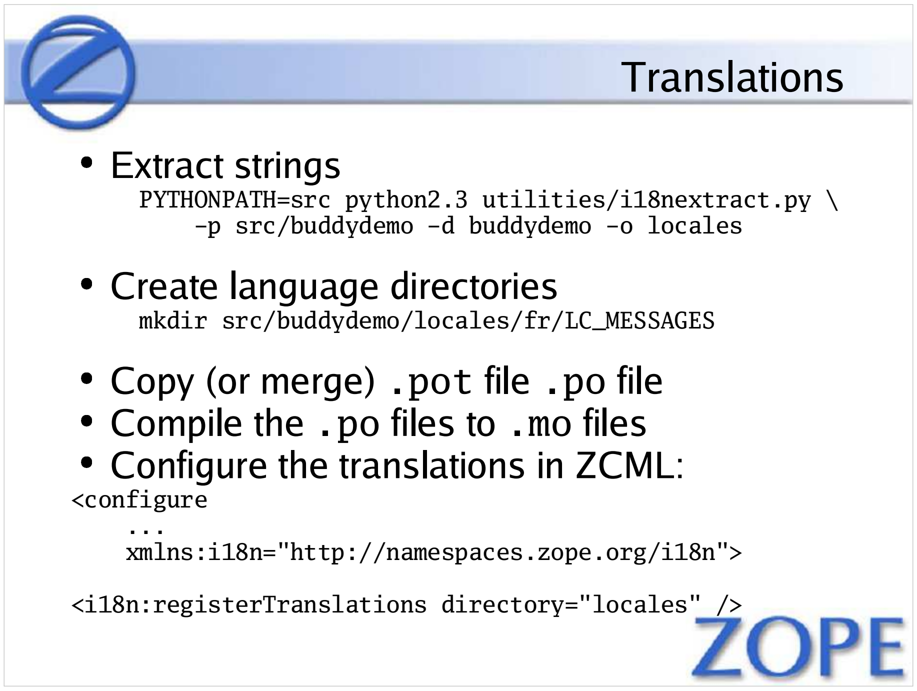
• “Data” are managed by content implementations directly

• “Meta data” are data managed by frameworks. Content implementations don't manage meta data. If they did, it wouldn't be meta data.

Note that the definitions of data and meta data are relative to an implementation. Imagine an automated library card catalog system. The system manages card catalog entries. Relative to the card catalog entries, the catalog entry information is data. Relative to the books in the library, they are meta data. The catalog system might keep track of when entries were updated. This information is meta-data relative to the entries.

Zope manages meta data using annotations. Annotations are managed independent of an object's implementation. The details of this framework are beyond the scope of this course.

We need to configure objects to indicate whether they are annotatable and how. Currently, we only support attribute annotations, which are stored in an `__annotations__` attribute on objects. We make an object attribute annotatable by declaring the `IAttributeAnnotatable` marker interface. Because this is primarily a configuration task, so we do it in ZCML.



Translations

- Extract strings

```
PYTHONPATH=src python2.3 utilities/i18nextract.py \
    -p src/buddydemo -d buddydemo -o locales
```

- Create language directories

```
mkdir src/buddydemo/locales/fr/LC_MESSAGES
```

- Copy (or merge) `.pot` file `.po` file
- Compile the `.po` files to `.mo` files
- Configure the translations in ZCML:

```
<configure
    ...
    xmlns:i18n="http://namespaces.zope.org/i18n">

<i18n:registerTranslations directory="locales" />
```

The extraction tool extracts translatable strings from Python, ZPT, and ZCML. You need to provide a path (`-p`) to a directory to be searches, a translation domain (`-d`) and the name of a directory (`-o`) to extract to. The output directory will be created if it doesn't exist.

The output directory will contain a file with a name equal to the translation domain and a ".pot" suffix. This is a translation template.

To create a translation for a particular language:

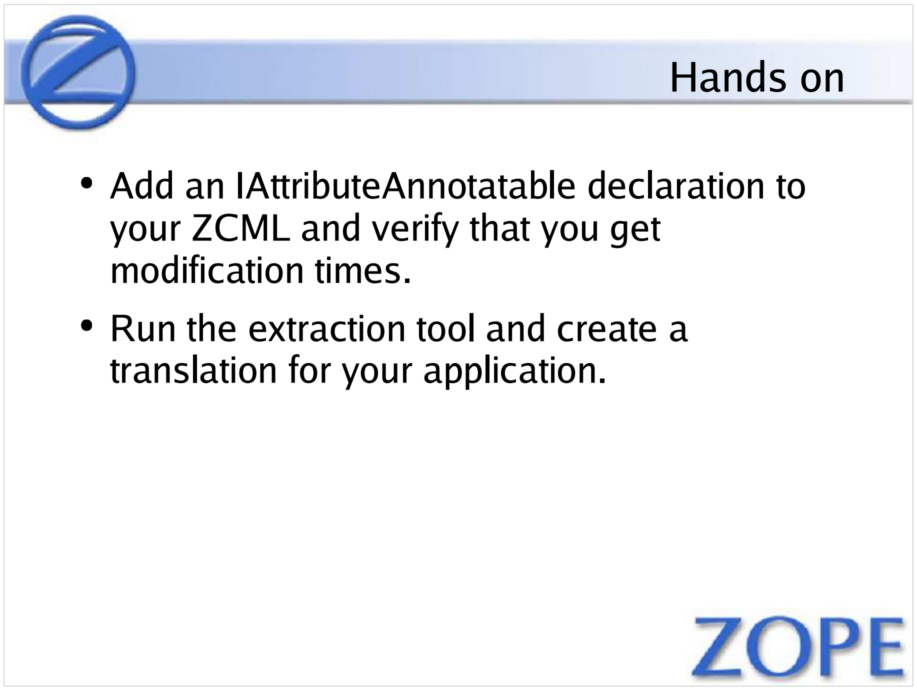
• Create a subdirectory of the translation directory who's name is a language code (e.g. "fr" or "en-us").

• Within the language directory, create the directory, LC_MESSAGES.

• Copy the template file to the LC_MESSAGES directory, but with a ".po" suffix. For example, a German translation of the buddydemo application would live in the file: `src/buddydemo/locales/de/LC_MESSAGES/buddydemo.po`. Later, when you update your software, you'll extract the strings into a new ".pot" file and use the gettext `merger's` tool to merge the ".pot" file changes into your ".po" file.

• Edit the ".po" file to add translations.

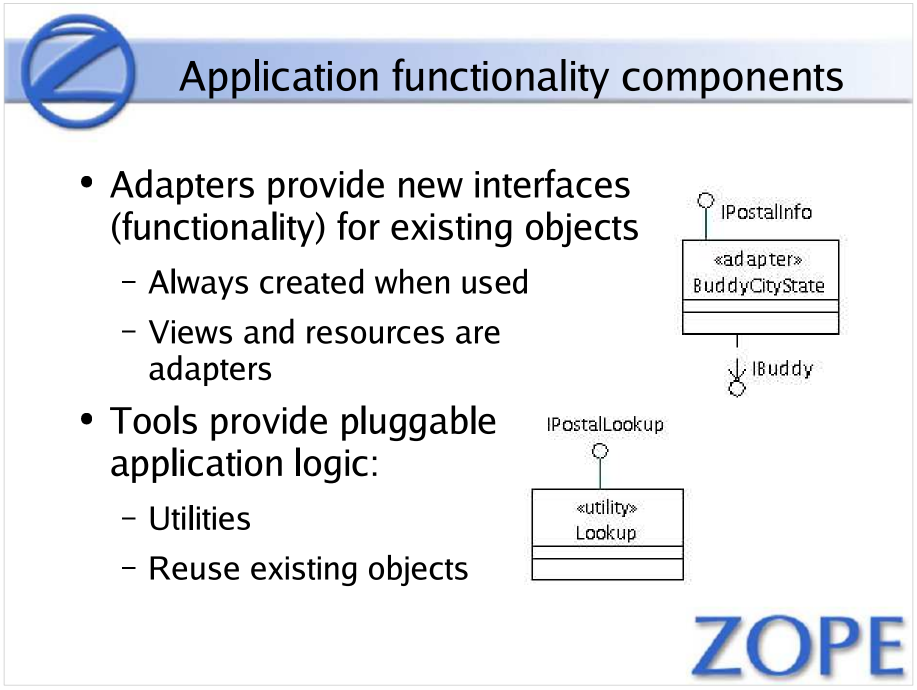
• Compile the translations using the gettext `msgfmt` tool.



Hands on

- Add an IAttributeAnnotatable declaration to your ZCML and verify that you get modification times.
- Run the extraction tool and create a translation for your application.

ZOPE



Application functionality components

- Adapters provide new interfaces (functionality) for existing objects
  - Always created when used
  - Views and resources are adapters
- Tools provide pluggable application logic:
  - Utilities
  - Reuse existing objects

IPostalInfo
«adapter»
BuddyCityState
IBuddy
IPostalLookup
«utility»
Lookup
ZOPE

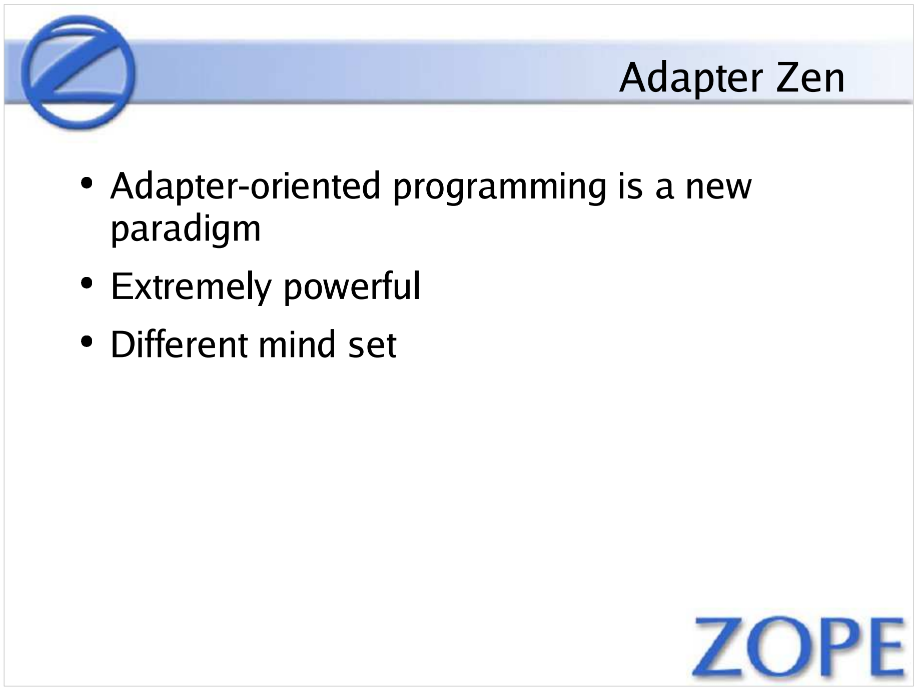
Simple adapters adapt one thing. Adapters can adapt zero, one, or many things. Views are actually multi-adapters that adapt an object (or multiple objects) and a request.

Whenever we ask the system for an adapter, a new object is created. Adapters are generally very transient. They are typically defined via classes with constructors that take the objects adapted as arguments.

Tools include utilities and services, but most, by far, are utilities. Tools are generally created once and used over and over. Multiple tool retrievals will return the same component. Tools are often persistent.

Utilities are registered by interface and name, although an empty string is often used as the name. We often register many utilities with the same interface. Database connections are registered as utilities implementing `IZopeDatabaseAdapter`. We may register several with different names and select which one to use in a context (e.g. an sql method).

Services are components that provide foundational functionality. Soon, we will have so few of these that we will stop talking about them.



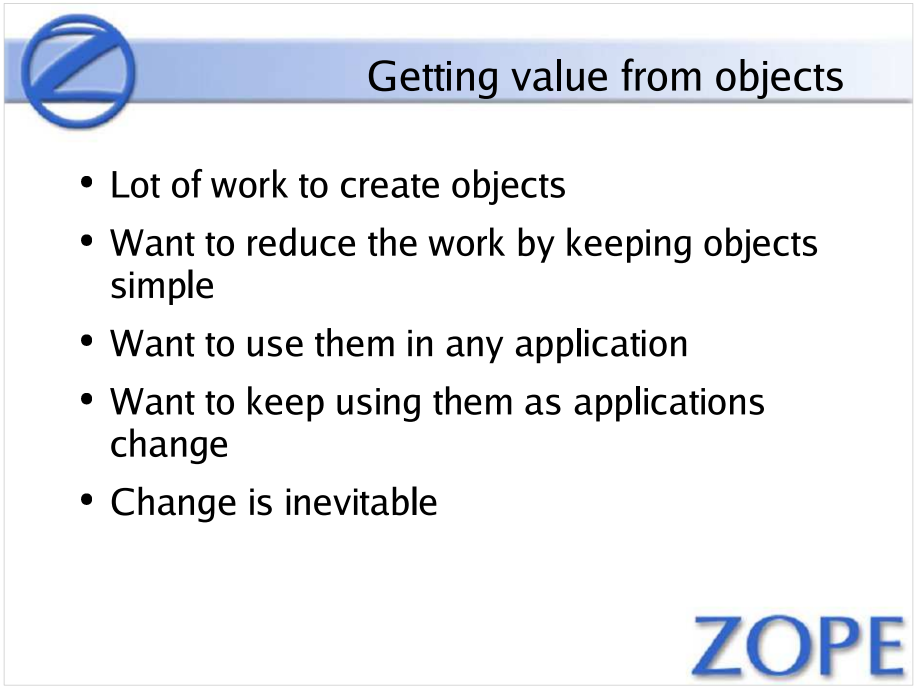
Adapter Zen

- Adapter-oriented programming is a new paradigm
- Extremely powerful
- Different mind set

ZOPE

Adapters provide a fundamentally different way to think about organizing software. Prepare yourself for a paradigm shift. The next few slides try to provide some motivation and introduction.

Ask lots of questions!



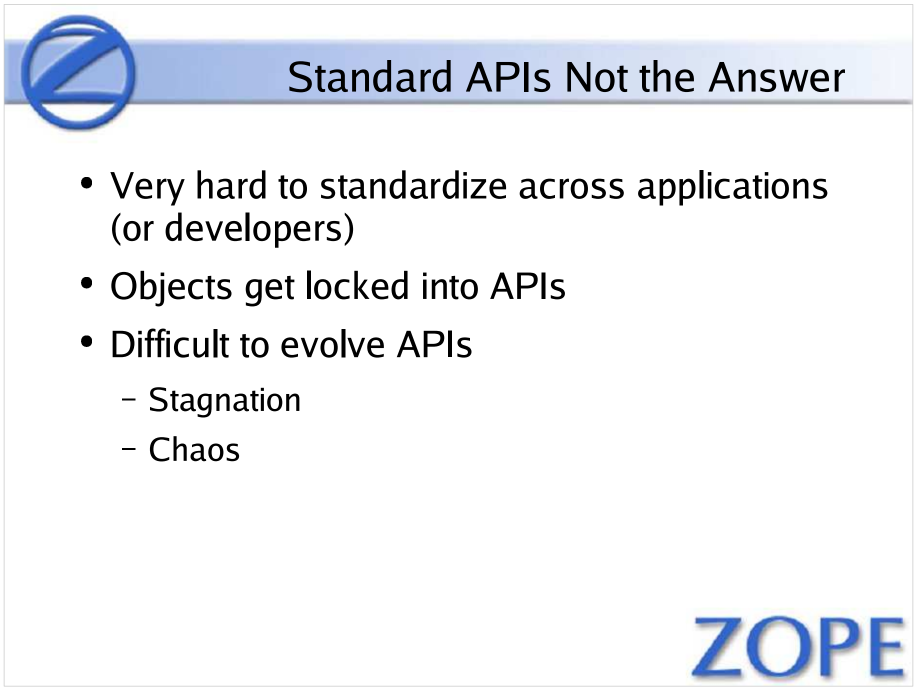
Getting value from objects

- Lot of work to create objects
- Want to reduce the work by keeping objects simple
- Want to use them in any application
- Want to keep using them as applications change
- Change is inevitable

We want to be able to reuse the software we create. Why? Software is expensive to create **and maintain**. This is even more so for quality software that has documentation and tests.

We also want to reduce the effort to create and maintain software by keeping the software as simple as we can. It's much easier to create, document, test and **understand** software that does **one** thing.

We want software components that survive change. This includes changing requirements and changing execution environments. We want to **minimize dependencies between components and their environment**.



Standard APIs Not the Answer

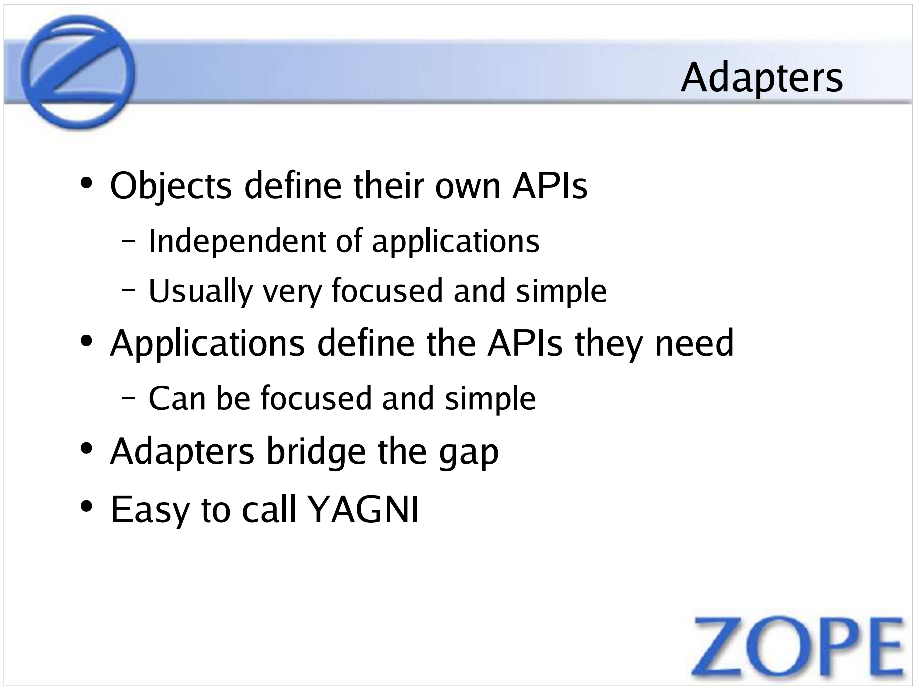
- Very hard to standardize across applications (or developers)
- Objects get locked into APIs
- Difficult to evolve APIs
  - Stagnation
  - Chaos

ZOPE

One way to reuse out objects across applications is through standard APIs. This rarely works.

It's usually too much work to achieve standardization. When we do, we often end up with lowest-common-denominator standards that don't meet the needs of applications or the framework.

When standards are achieved, it becomes hard to change to reflect new requirements. Either standards don't change to meet changing needs, or an evolving standards and applications that use them have to deal with components supporting different versions of the standard.



Adapters

- Objects define their own APIs
  - Independent of applications
  - Usually very focused and simple
- Applications define the APIs they need
  - Can be focused and simple
- Adapters bridge the gap
- Easy to call YAGNI

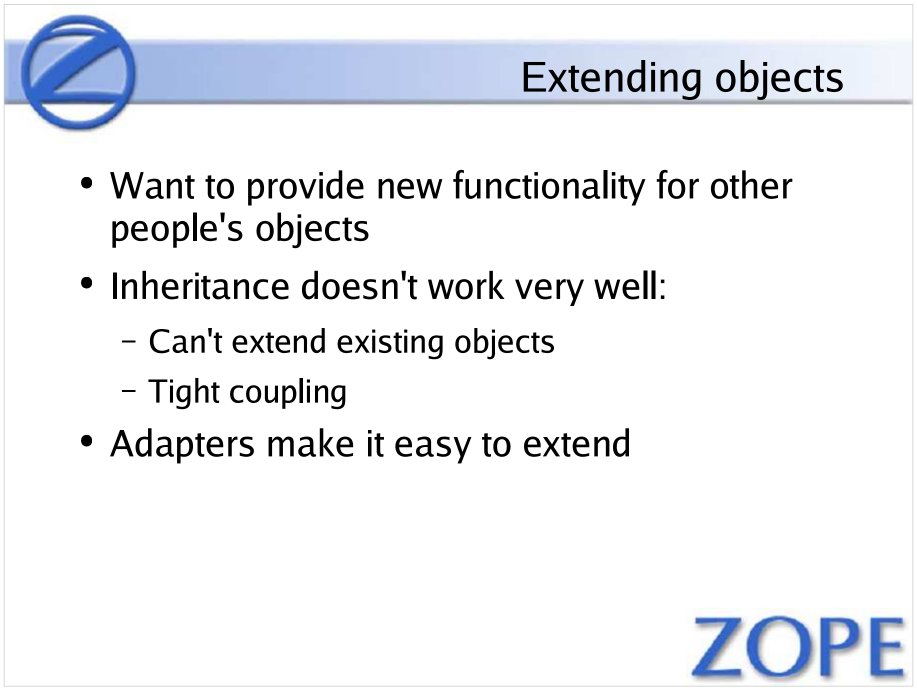
ZOPE

With adapters (and interfaces), components do just what they need to do. They don't need to anticipate possible framework requirements.

Likewise, frameworks can require only what they need. They can change what they need fairly easily, so they don't have to anticipate future needs.

Adapters take care of translating between APIs. Changing APIs isn't painless, but it can, at least, be controlled through adaptation.

APIs **don't have to include features they don't need but might need later**. API designers can call YAGNI. If their wrong, change is straightforward.



Extending objects

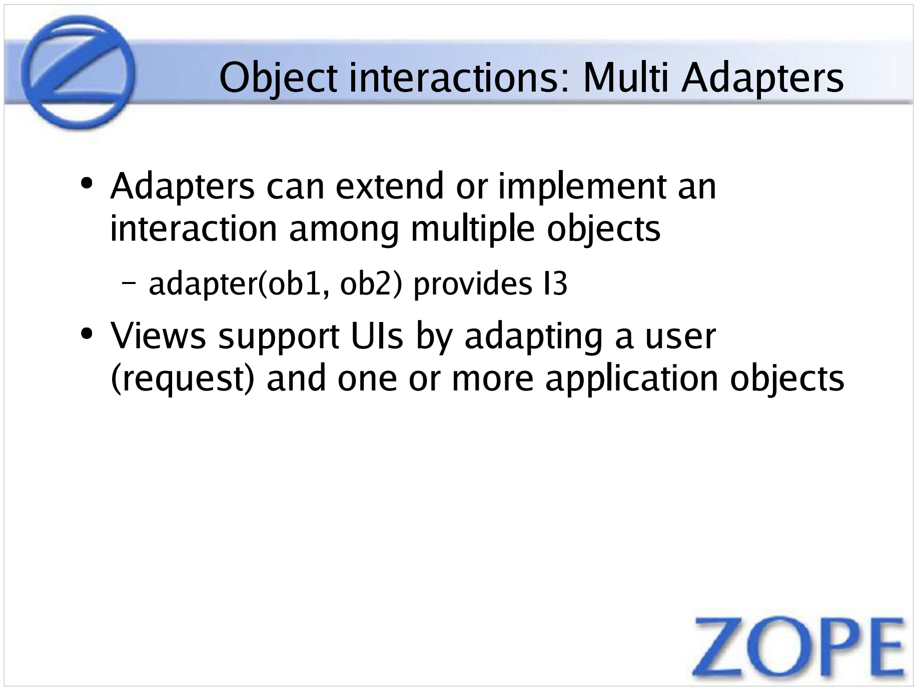
- Want to provide new functionality for other people's objects
- Inheritance doesn't work very well:
  - Can't extend existing objects
  - Tight coupling
- Adapters make it easy to extend

You get a useful component from someone, but it lacks some feature you need. Maybe you want to add some operations, or perhaps you need to give it a user interface. Perhaps it just needs to have an API that makes it fit your application.

You could modify the component source, but that will cause a maintenance head ache. Every time you get a new version of the component from it's author, you'll have to redo your modifications.

You could subclass the component's class and create your own version. Your new version of the component will be tightly coupled to the components implementation. New versions of the component could change their implementations in ways that break your subclass.

If you use persistent objects, you have another problem with subclassing. Any persistent instances of the component won't get any of your new features. You either need to live with that, or write a conversion script that replaces instances of the old component with instances of the subclass.

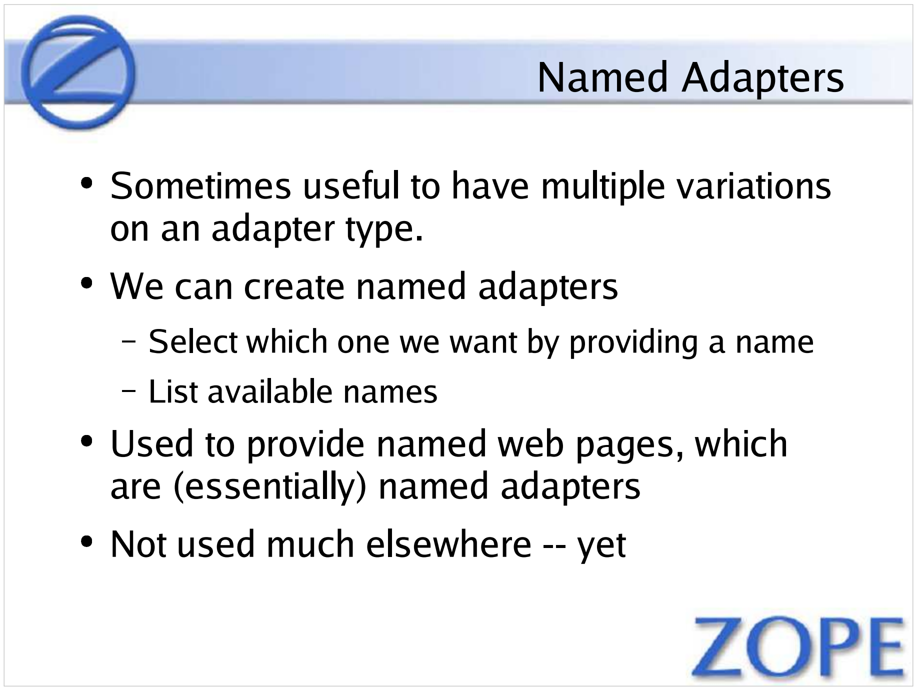


Object interactions: Multi Adapters

- Adapters can extend or implement an interaction among multiple objects
  - adapter(ob1, ob2) provides I3
- Views support UIs by adapting a user (request) and one or more application objects

Sometimes you want to implement an application feature that depends on multiple objects. The most common example of this is a user interface, which involves some application object(s) and a user, where the user is represented by some object in the system, a request in Zope. We use multi-adapters for this. Like simple adapters, multi-adapters are defined by factories, but unlike simple adapters, we don't check for __conform__ methods on the objects being adapted or whether an object already provides the interface.

Of course, multi-adapter factories are passed arguments for each of the objects being adapted.



Named Adapters

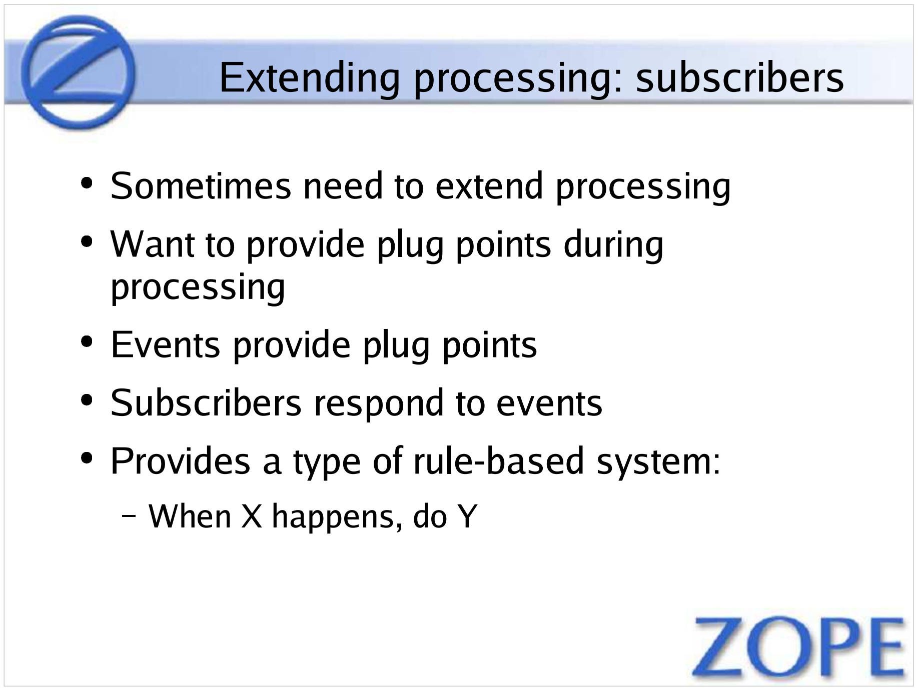
- Sometimes useful to have multiple variations on an adapter type.
- We can create named adapters
  - Select which one we want by providing a name
  - List available names
- Used to provide named web pages, which are (essentially) named adapters
- Not used much elsewhere -- yet

ZOPE

Named adapters let you have multiple versions of the same basic adapter type, where an adapter type a combination of a provided type and zero or more adapted types.

Note that, as with multi-adapters, an adapted objects __conform__ method and provided interfaces are not considered when looking up named adapters.

Named adapters can be single or multi-adapters.



Extending processing: subscribers

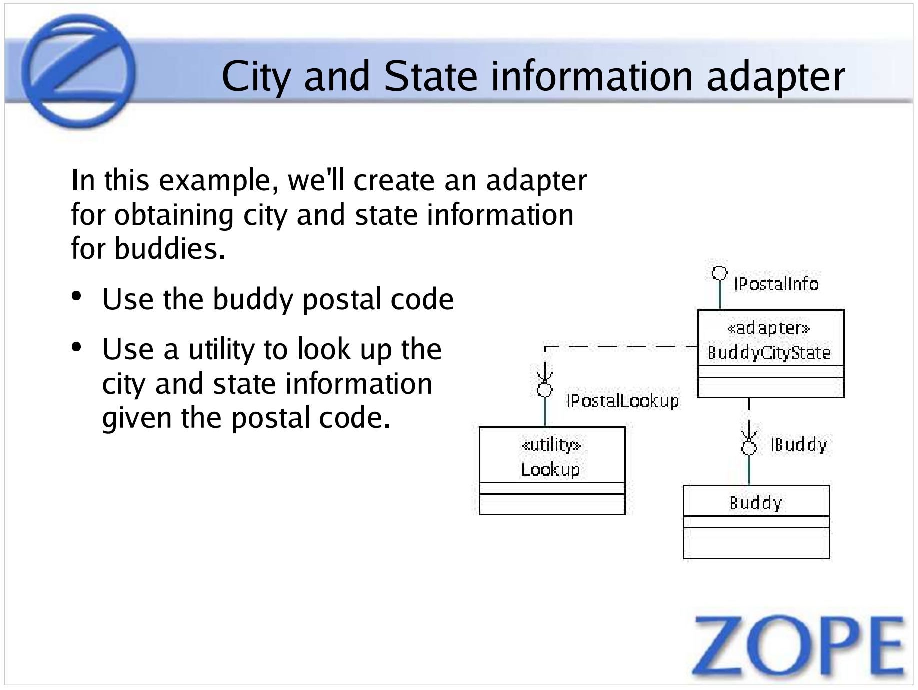
- Sometimes need to extend processing
- Want to provide plug points during processing
- Events provide plug points
- Subscribers respond to events
- Provides a type of rule-based system:
  - When X happens, do Y

ZOPE

Sometimes, rather than providing or replacing functionality, we want to extend existing functionality. One way to do this is to define points in normal processing where extra processing could occur. At these points, we notify event subscribers.

Subscribers are another kind of adapters. They differ from other adapters in the way that they are registered and looked up. With non-subscribers, we register a single adapter for an adapter type (required and provided interfaces) and name. When you look up a non-subscriber adapter, you get a single adapter back (if any) that represents the best fit for the object being adapted and the desired interface. You can register multiple subscription adapters for the same adapter type and, when you look up subscribers, you get all that match.

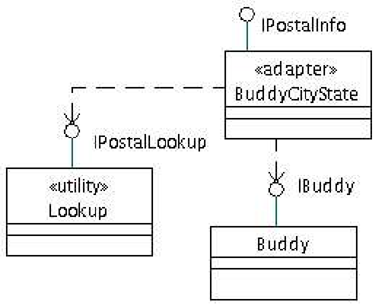
There's a special case for subscriptions. Normally, creating adapters doesn't have any side effects. For subscribers, we often register functions as subscribers that return None. In this case, the factories do all of the work, which usually involves side effects. This special case is vary valuable, as it allows event subscribers to be implemented by simple functions, rather than classes.



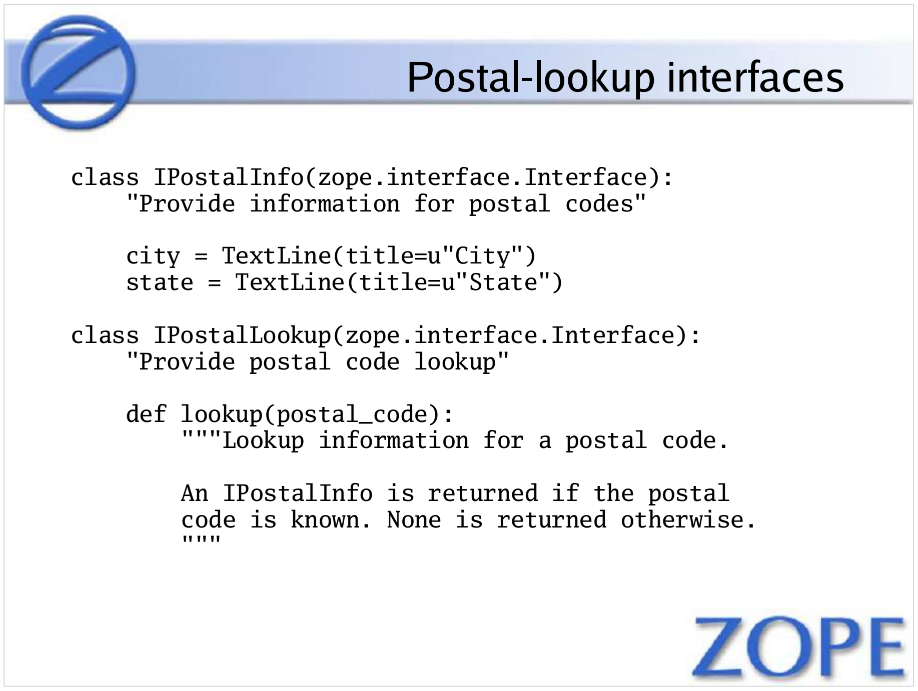
City and State information adapter

In this example, we'll create an adapter for obtaining city and state information for buddies.

- Use the buddy postal code
- Use a utility to look up the city and state information given the postal code.



The buddy example is contrived to provide an opportunity to use an adapter and a utility. We didn't include a city and state in the buddy data because we can compute them from the postal code. To do this, we need to be able to lookup a city and state given a postal code. It seems likely that one would want to do such lookup in other places, so we factor the lookup into a separate utility.

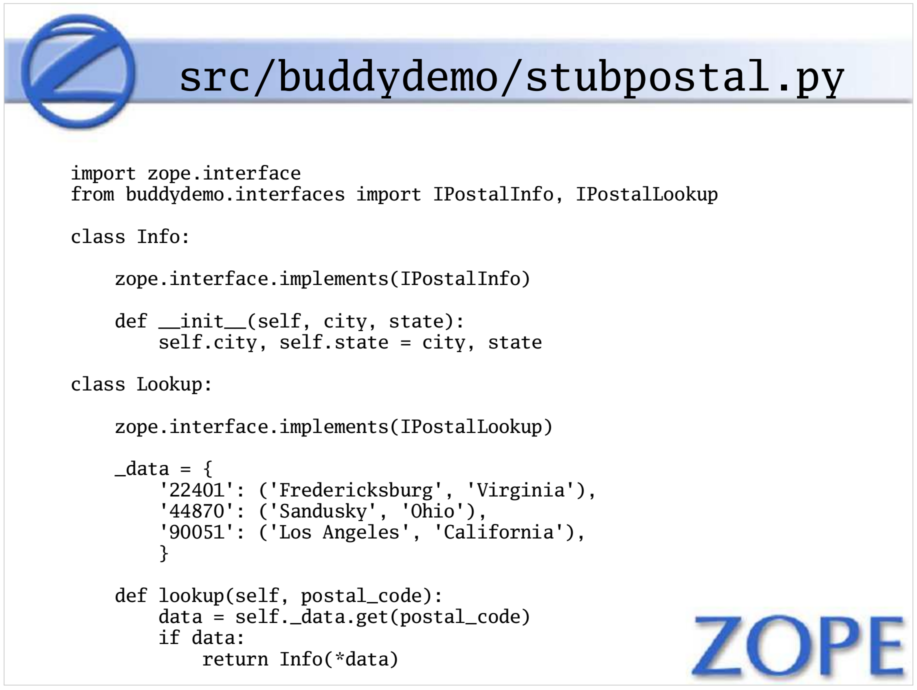


Postal-lookup interfaces

```
class IPostalInfo(zope.interface.Interface):
    "Provide information for postal codes"

    city = TextLine(title=u"City")
    state = TextLine(title=u"State")

class IPostalLookup(zope.interface.Interface):
    "Provide postal code lookup"

    def lookup(postal_code):
        """Lookup information for a postal code.

        An IPostalInfo is returned if the postal
        code is known. None is returned otherwise.
        """
```

Our utility will implement `IPostalLookup`.

`IPostalInfo` is used to describe the data returned. `IPostalInfo` is an example of a non-persistent content-component interface.



src/buddydemo/stubpostal.py

```
import zope.interface
from buddydemo.interfaces import IPostalInfo, IPostalLookup

class Info:

    zope.interface.implements(IPostalInfo)

    def __init__(self, city, state):
        self.city, self.state = city, state

class Lookup:

    zope.interface.implements(IPostalLookup)

    _data = {
        '22401': ('Fredericksburg', 'Virginia'),
        '44870': ('Sandusky', 'Ohio'),
        '90051': ('Los Angeles', 'California'),
        }

    def lookup(self, postal_code):
        data = self._data.get(postal_code)
        if data:
            return Info(*data)
```

A proper postal lookup utility implementation will have use some sort of database for looking up postal-code data. We don't want to bother will that now, so we create a temporary stub implementation later.

One of the main benefits of utilities is that implementations can be swapped out without affecting clients.

Info object provide basic information storage for a city and state:

```
  >>> from buddydemo import stubpostal
  >>> info = stubpostal.Info('Cleveland', 'Ohio')
  >>> info.city, info.state
  ('Cleveland', 'Ohio')
```

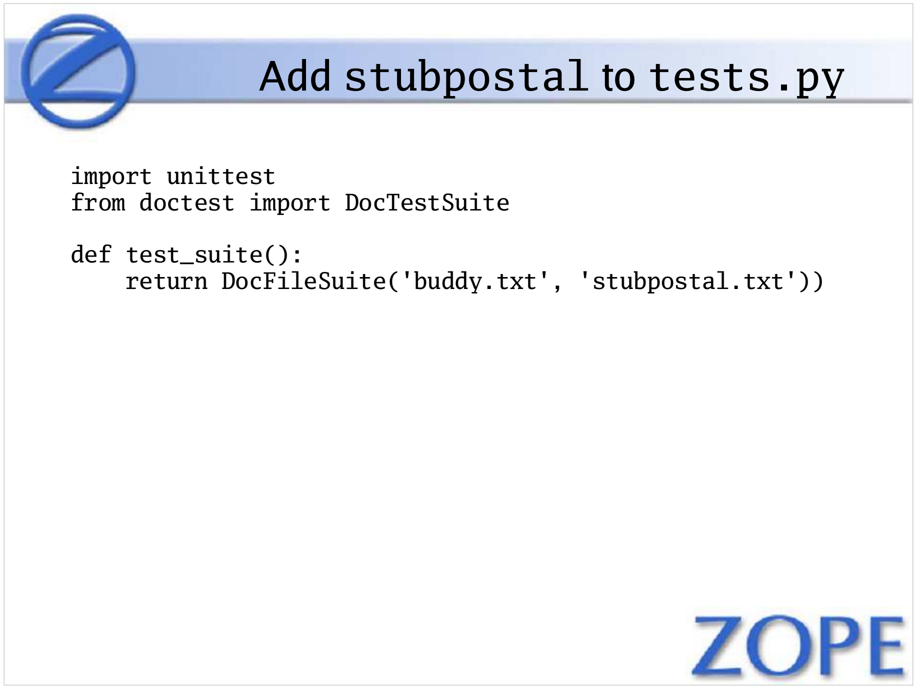
Lookup objects provide an IPostalLookup implementation with a small test database.

If a known postal code is used, we get data:

```
  >>> lookup = stubpostal.Lookup()
  >>> info = lookup.lookup('22401')
  >>> info.city, info.state
  ('Fredericksburg', 'Virginia')
```

But get nothing if an unknown code is given:

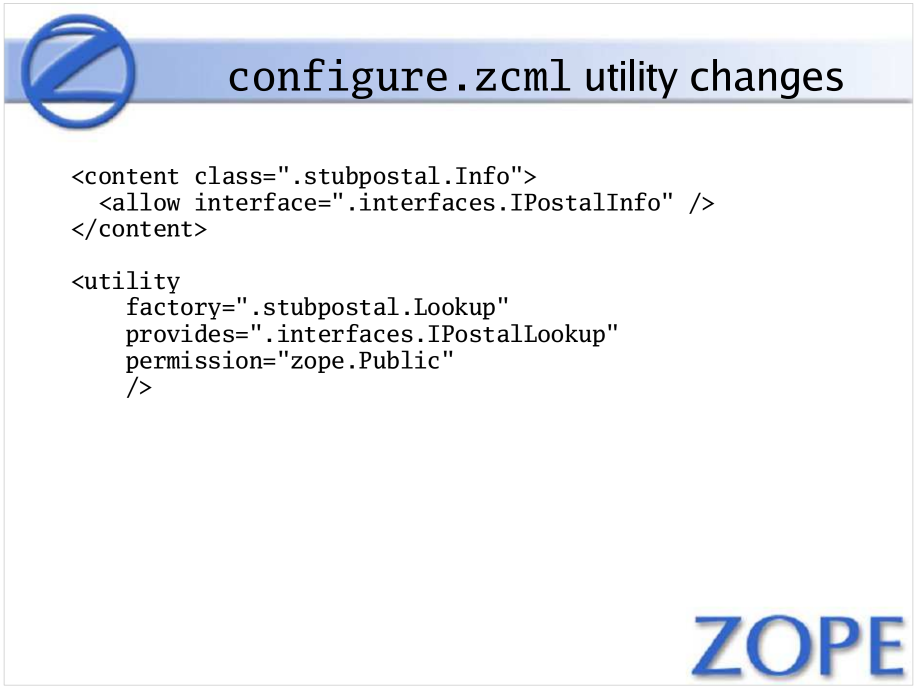
```
  >>> lookup.lookup('11111')
```



Add stubpostal to tests.py

```
import unittest
from doctest import DocTestSuite

def test_suite():
    return DocFileSuite('buddy.txt', 'stubpostal.txt'))
```

Note that the notes for the previous slide contain most of the text of `stubpostal.txt`.



configure.zcml utility changes

```
<content class=".stubpostal.Info">
  <allow interface=".interfaces.IPostalInfo" />
</content>

<utility
    factory=".stubpostal.Lookup"
    provides=".interfaces.IPostalLookup"
    permission="zope.Public"
    />
```

As mentioned earlier, `IPostalInfo` is a content type. We need to provide security declarations so that we can access it's methods. We use a new security-declaration directive, `allow`. The `allow` directive declares that access to an interface or attributes is **always** allowed.

We use a `utility` directive to register our utility. We specify a factory for creating the utility. We could, instead, specify an existing utility instance using a `component` attribute. This would be necessary if we needed to supply data to initialize the component. In this case, using the `factory` attribute is more convenient, since it allows us to avoid creating an instance in our Python module.

We specify what interface the utility provides using the `provides` attribute. We could also supply a name. In this example, we accept the default name, which is an empty string.

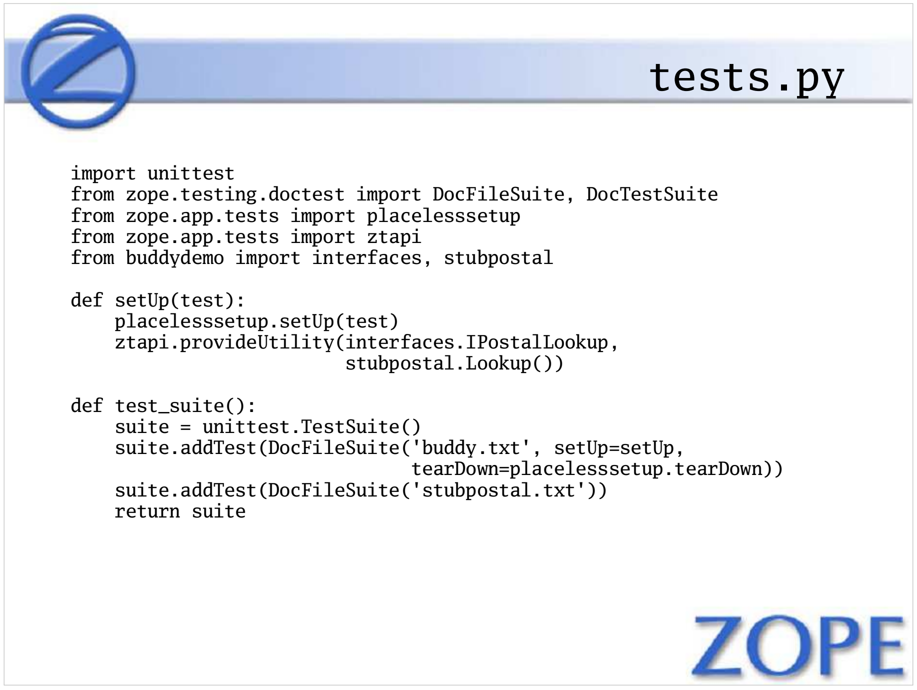
We specify the permission necessary to use the utility. The permission is optional, however, if it isn't set, then untrusted code will be unable to use the utility.

In the utility definition, we used the special permission, `zope.Public`. The `zope.Public` permission is special because it provides unconditional access. Anything that requires `zope.Public` is always unconditionally available.

As you might have guessed, the `allow` directive is equivalent to the `require` directive with a permission of `zope.Public`.

More tests in `buddy.txt`

```
Getting City and State Information
----------------------------------

We can get city and state information for a buddy using the buddy
city-state adapter:

  >>> from buddydemo.buddy import BuddyCityState
  >>> bob = Buddy('Bob', 'Smith', 'bob@zope.com', '4 foo street',
  ...             '22401')
  >>> info = BuddyCityState(bob)
  >>> info.city, info.state
  ('Fredericksburg', 'Virginia')

The city state adapter provides empty strings of the postal code is
omitted or not in whatever database is provided by a postal-lookup
utility:

  >>> info = BuddyCityState(Buddy('Bob'))
  >>> info.city, info.state
  ('', '')

  >>> info = BuddyCityState(Buddy('Bob', pc='11111'))
  >>> info.city, info.state
  ('', '')
```



tests.py

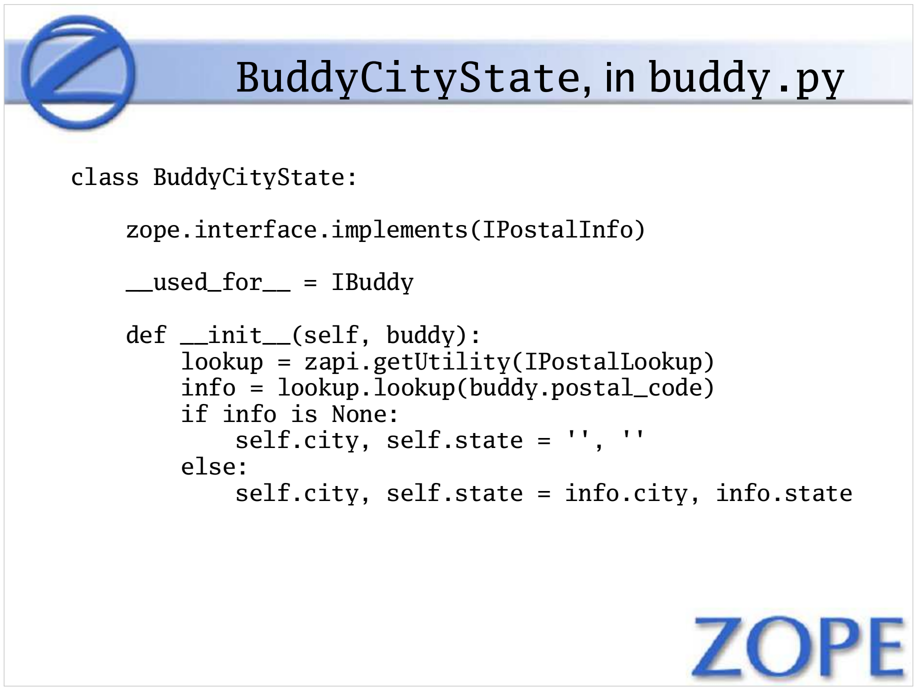
```
import unittest
from zope.testing.doctest import DocFileSuite, DocTestSuite
from zope.app.tests import placelesssetup
from zope.app.tests import ztapi
from buddydemo import interfaces, stubpostal

def setUp(test):
    placelesssetup.setUp(test)
    ztapi.provideUtility(interfaces.IPostalLookup,
                         stubpostal.Lookup())

def test_suite():
    suite = unittest.TestSuite()
    suite.addTest(DocFileSuite('buddy.txt', setUp=setUp,
                               tearDown=placelesssetup.tearDown))
    suite.addTest(DocFileSuite('stubpostal.txt'))
    return suite
```

Because our adapter is going to look up a utility, we need to arrange for the component architecture to be initialized. We do this with zope.app.tests.placelesssetup.setUp.

Once that's done, we need to register a stub utility. We use helper function zope.app.tests.ztapi.provideUtility. The module zope.app.tests.ztapi has a number of methods that help set up components for tests.

We do all of this in a setUp function. zope.app.tests.placelesssetup defines setUp and tearDown functions that set up and tear down the basic component environment. We can use the tearDown method as is, but we need to augment the setUp function to register our stub utility.

The setUp and tearDown functions take an argument, which is a doctest.DocTest object. This is passed to allow us to get to the test "globals" (`test.globs`). This is useful for some advanced situations. We don't need it here.



BuddyCityState, in buddy.py

```
class BuddyCityState:

    zope.interface.implements(IPostalInfo)

    __used_for__ = IBuddy

    def __init__(self, buddy):
        lookup = zapi.getUtility(IPostalLookup)
        info = lookup.lookup(buddy.postal_code)
        if info is None:
            self.city, self.state = '', ''
        else:
            self.city, self.state = info.city, info.state
```

Our adapter implements IPostalInfo.

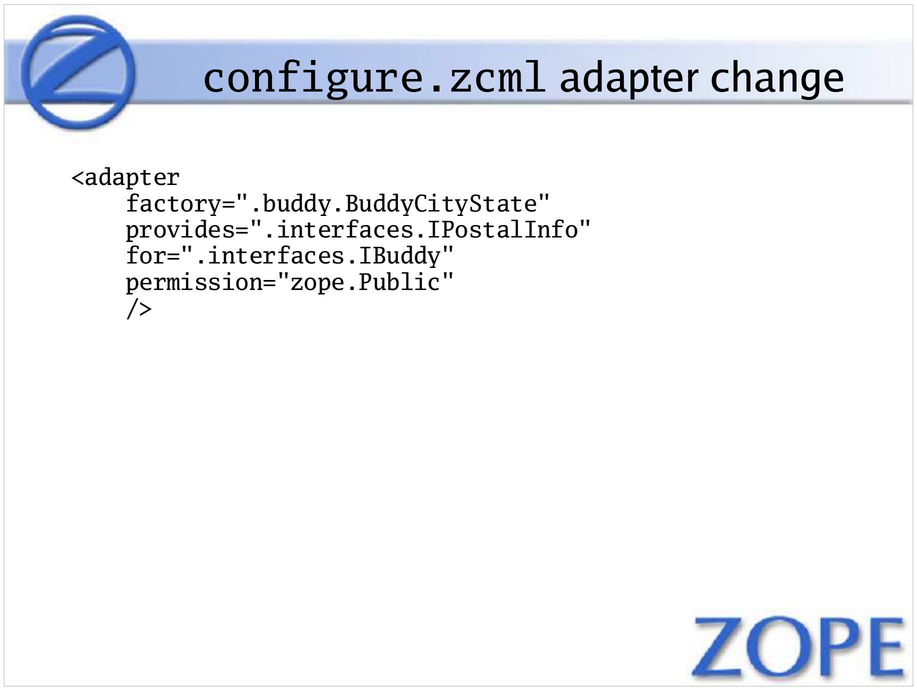
We set a __used_for__ attribute to document the interface we depend on. This is purely a documentation convention at this point. In the future, we might use this to check configurations.

The constructor takes the object to be adapted, which is an IBuddy. In this example, all of the work is done in the constructor.

We look up a utility by calling zapi.getUtility. The zapi module is a convenience module that gathers together a variety of widely used application programming interfaces. See zope/app/interfaces/zapi.py (and the modules it imports) for details.

The getUtility method raises an exception if a utility can't be found. Generally "get" methods raise errors if they can't find something. There are usually "query" methods (e.g. queryUtility) that return a default value (defaulting to None) if a value can't be found. The first argument to getUtility is an object that provides a place to look up a utility. Generally, components can be defined locally to a site. When we look something up, we provide a location to look for the component in. This is a form of acquisition. The second argument to getUtility specifies the desired interface. A name may also be provided as an additional argument or as a name keyword argument, and defaults to an empty string.

The constructor uses the utility to look up postal information, saving the information away for later use.

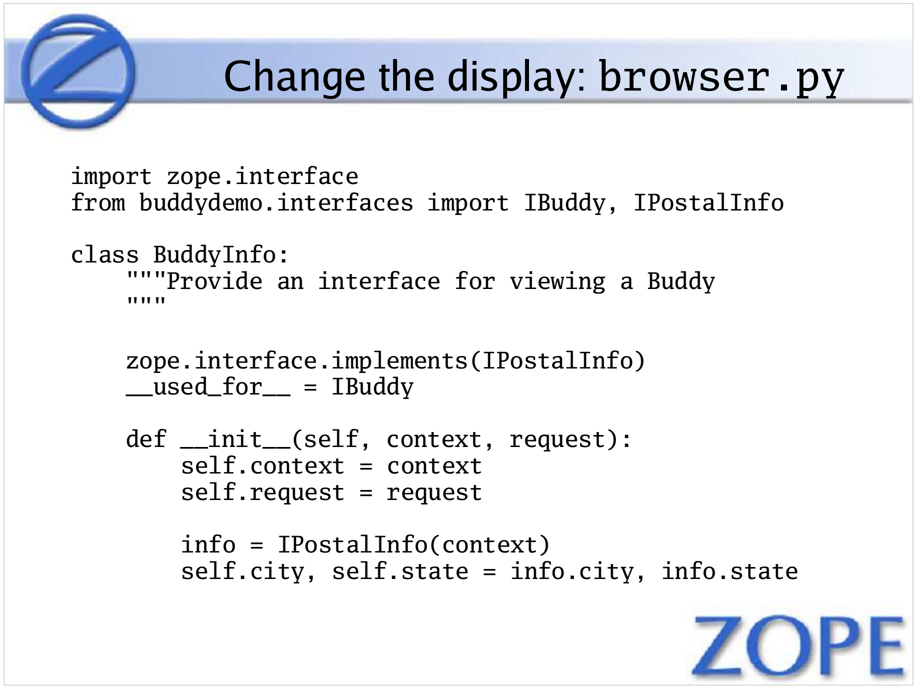


configure.zcml adapter change

```
<adapter
    factory=".buddy.BuddyCityState"
    provides=".interfaces.IPostalInfo"
    for=".interfaces.IBuddy"
    permission="zope.Public"
    />
```

The adapter directive is similar to the utility directive. Like the utility directive, it specifies the provided interface, a factory and a permission. As with utilities, a name may be provided.

The adapter directive also requires the use of a for attribute, which specifies the interface the adapter is used for. You can specify that an adapter is for all objects by supplying and asterisk (*) for the interface. You can specify many interfaces, separated by spaces, or you can specify no interfaces.

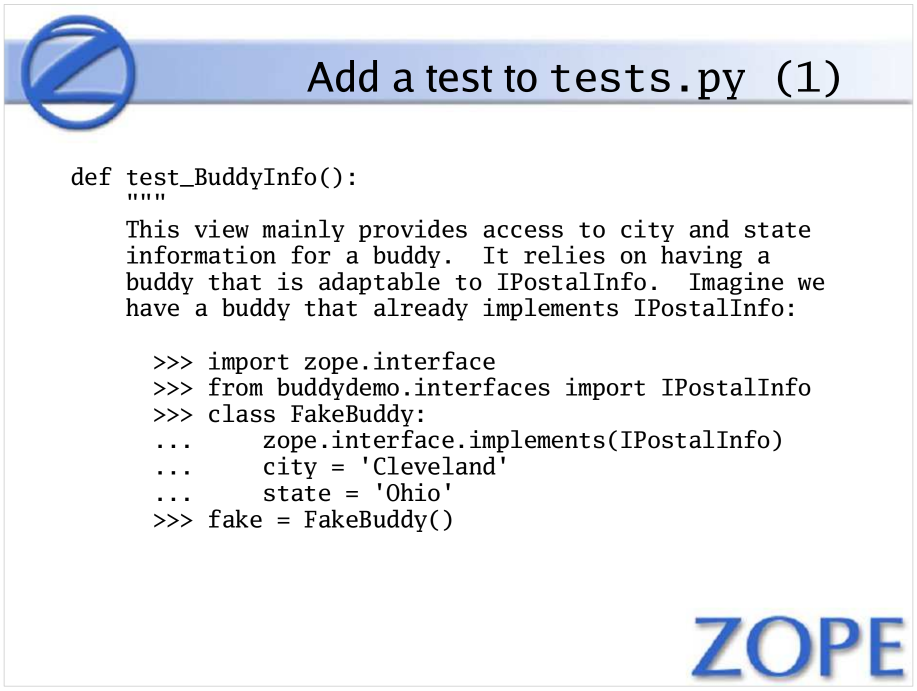


Change the display: browser.py

```
import zope.interface
from buddydemo.interfaces import IBuddy, IPostalInfo

class BuddyInfo:
    """Provide an interface for viewing a Buddy
    """

    zope.interface.implements(IPostalInfo)
    __used_for__ = IBuddy

    def __init__(self, context, request):
        self.context = context
        self.request = request

        info = IPostalInfo(context)
        self.city, self.state = info.city, info.state
```

Now that we have a way to get a city and state, we can improve our display view to include the city and state. We'll create a view class that provides city and state attributes to be used by our ZPT template. The view uses the adapter in the constructor to get the city and state information.

We get an adapter by just calling the interface. If the object passed to the interface already implements it, then the object will be returned.

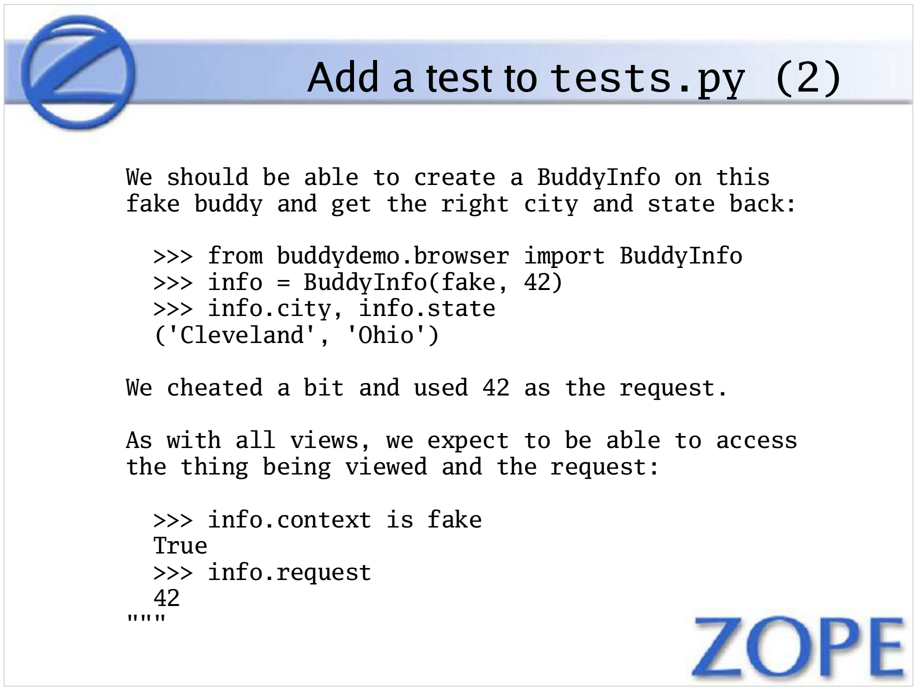


Add a test to `tests.py (1)`

```
def test_BuddyInfo():
    """
    This view mainly provides access to city and state
    information for a buddy.  It relies on having a
    buddy that is adaptable to IPostalInfo.  Imagine we
    have a buddy that already implements IPostalInfo:

      >>> import zope.interface
      >>> from buddydemo.interfaces import IPostalInfo
      >>> class FakeBuddy:
      ...     zope.interface.implements(IPostalInfo)
      ...     city = 'Cleveland'
      ...     state = 'Ohio'
      >>> fake = FakeBuddy()
```

In this case, we decided to put the test in the test file. To do that, we just put a function in the test file with the desired doc string.

To test the view, we need an object that can be adapted to IPostalInfo. We could set up an adapter, but then we'd need to set up the component architecture and register the adapter. An easier way to accomplish this is to pass a stub object that already implements the desired interface. Then the adapter retrieval (interface call) will just return the stub object,

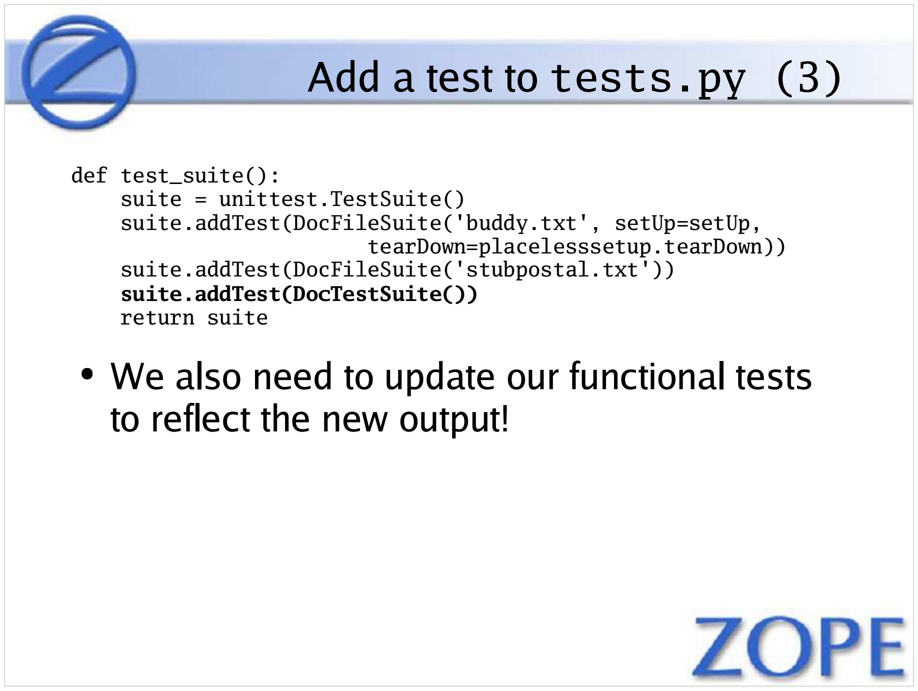


Add a test to `tests.py (2)`

```
We should be able to create a BuddyInfo on this
fake buddy and get the right city and state back:

  >>> from buddydemo.browser import BuddyInfo
  >>> info = BuddyInfo(fake, 42)
  >>> info.city, info.state
  ('Cleveland', 'Ohio')

We cheated a bit and used 42 as the request.

As with all views, we expect to be able to access
the thing being viewed and the request:

  >>> info.context is fake
  True
  >>> info.request
  42
"""
```

We normally have to pass a request to a view. We know that this view ignores it's request, so we just pass 42.

A view that uses ZPT is required to expose “context” and “request” attributes. These are needed so that the ZPT template can expose the information as ZPT `context` and `view` top-level names.


Add a test to `tests.py (3)`

```
def test_suite():
    suite = unittest.TestSuite()
    suite.addTest(DocFileSuite('buddy.txt', setUp=setUp,
                               tearDown=placelesssetup.tearDown))
    suite.addTest(DocFileSuite('stubpostal.txt'))
    suite.addTest(DocTestSuite())
    return suite
```

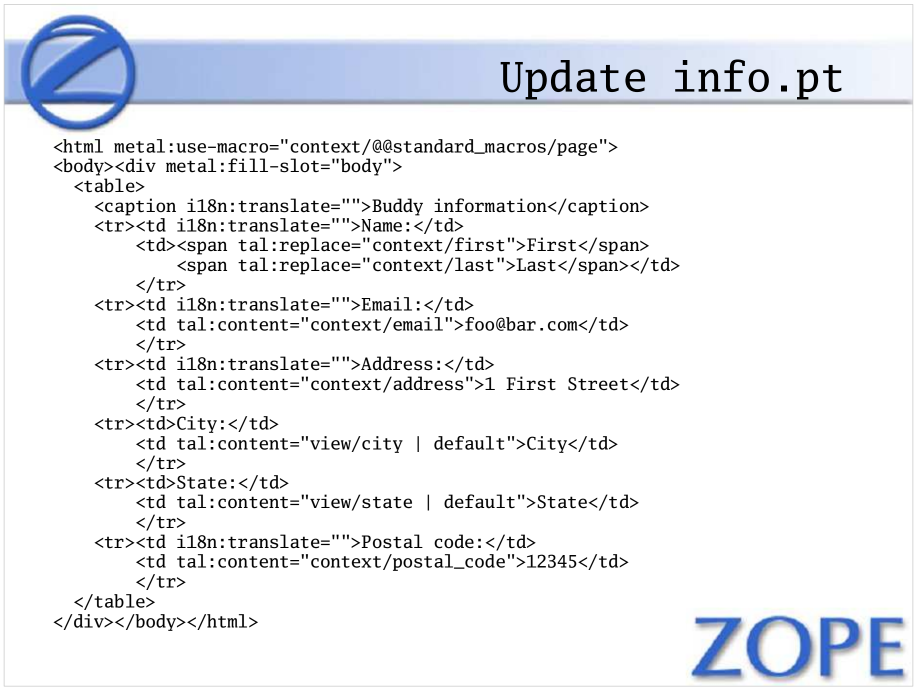
- We also need to update our functional tests to reflect the new output!

ZOPE

When the tests are in the test module, we need to tell `DocTestSuite` to look for tests in the calling module. We do that by calling `DocTestSuite` without an argument.

It can be very useful to write unit tests for the Python code in views. This is especially true if the Python code is complicated. It's much easier to debug code in a unit test that in the web server, or even in a functional test.

Given that we need to create a functional test anyway, however, I would normally not bother to write tests for very simple code like the code we have here, as long as the functional test exercises the code.

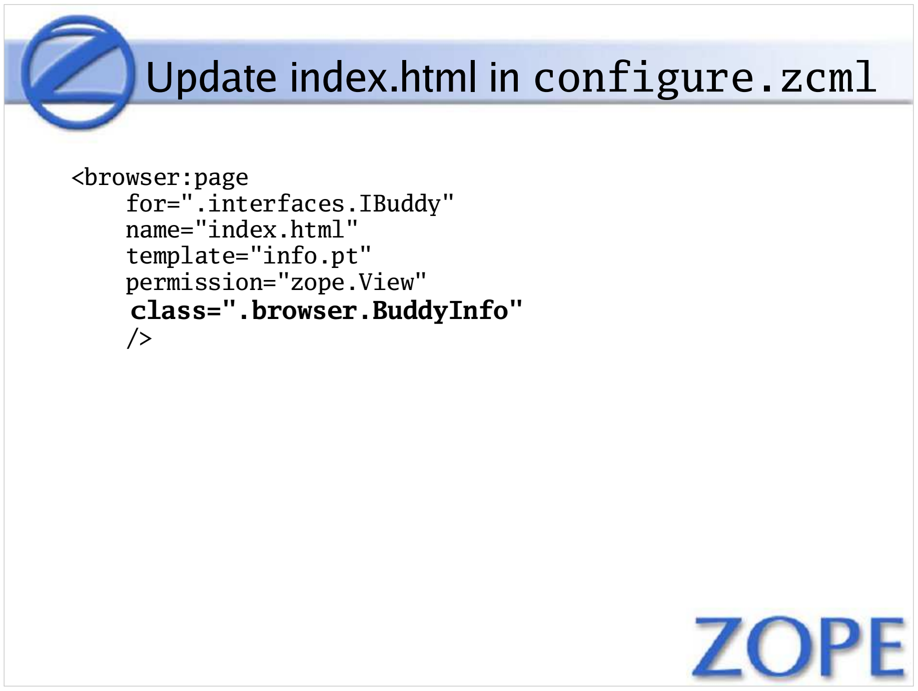


Update info.pt

```
<html metal:use-macro="context/@@standard_macros/page">
<body><div metal:fill-slot="body">
  <table>
    <caption i18n:translate="">Buddy information</caption>
    <tr><td i18n:translate="">Name:</td>
        <td><span tal:replace="context/first">First</span>
            <span tal:replace="context/last">Last</span></td>
        </tr>
    <tr><td i18n:translate="">Email:</td>
        <td tal:content="context/email">foo@bar.com</td>
        </tr>
    <tr><td i18n:translate="">Address:</td>
        <td tal:content="context/address">1 First Street</td>
        </tr>
    <tr><td>City:</td>
        <td tal:content="view/city | default">City</td>
        </tr>
    <tr><td>State:</td>
        <td tal:content="view/state | default">State</td>
        </tr>
    <tr><td i18n:translate="">Postal code:</td>
        <td tal:content="context/postal_code">12345</td>
        </tr>
  </table>
</div></body></html>
```

We update the display template to include the city and state.

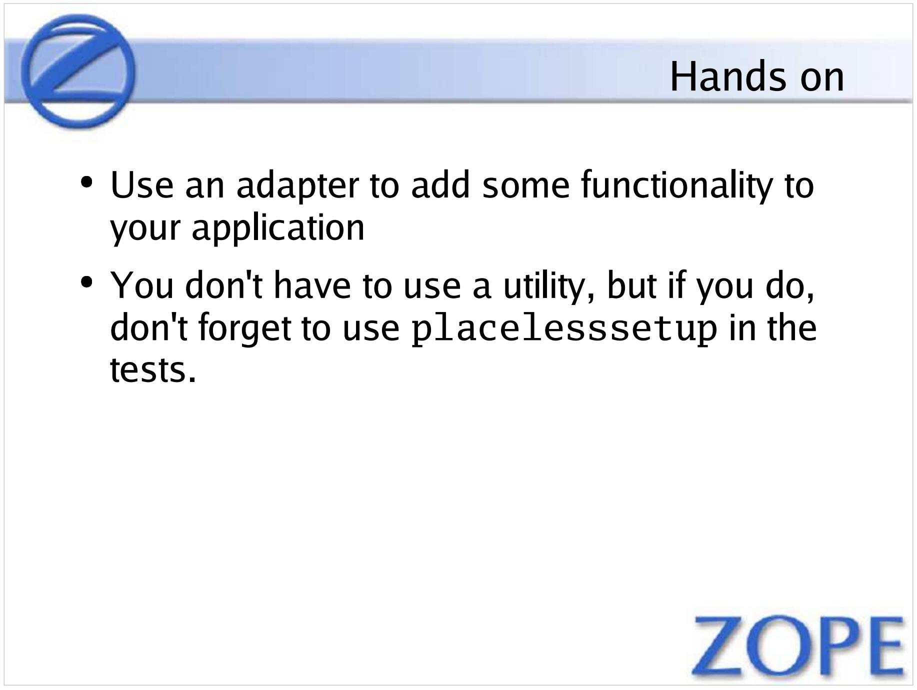
Note that we use the `view` variable to refer to the view and get the view's attributes, `city` and `state`. When a ZPT template is used in a view, it has a `view` top-level variable to provide access to the view.



Update index.html in configure.zcml

```
<browser:page
    for=".interfaces.IBuddy"
    name="index.html"
    template="info.pt"
    permission="zope.View"
    class=".browser.BuddyInfo"
    />
```

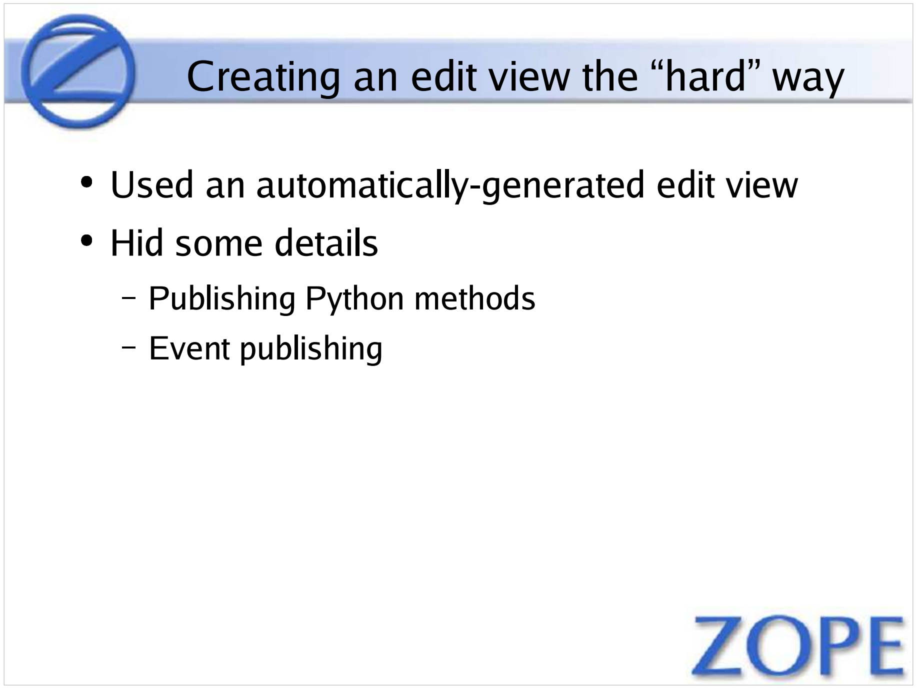
We update the page directive to include a class. The class is used as a mix- in class for the view.



Hands on

- Use an adapter to add some functionality to your application
- You don't have to use a utility, but if you do, don't forget to use `placelesssetup` in the tests.

ZOPE



Creating an edit view the “hard” way

- Used an automatically-generated edit view
- Hid some details
  - Publishing Python methods
  - Event publishing

ZOPE

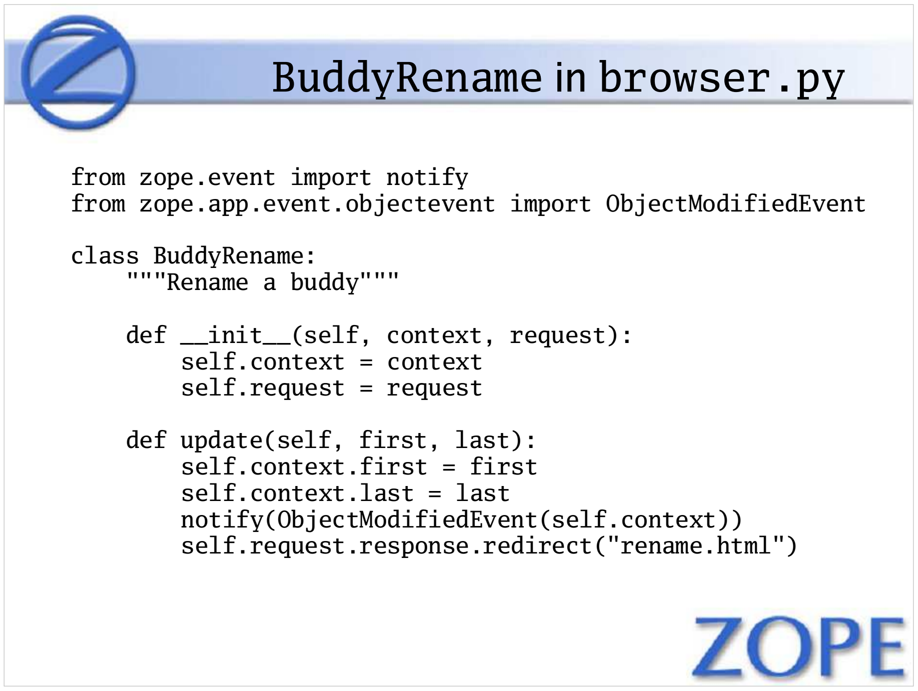
We took advantage of schemas to avoid most of the drudgery of creating edit and add views. This caused us to miss some important concepts.

As an example, we'll create a “rename” view that lets us enter first and last names.

src/buddydemo/rename.pt

```
<html metal:use-macro="context/@@standard_macros/page"
      i18n:domain="buddydemo">
<body><div metal:fill-slot="body">
<p i18n:translate="">Enter the Buddy information</p>
<form action="renameAction.html" method="post">
<table>
  <tr><td i18n:translate="">First name</td>
      <td><input type="text" name="first" size="40" value=""
           tal:attributes="value context/first" /> </td>
  </tr>
  <tr><td i18n:translate="">Last name</td>
      <td><input type="text" name="last" size="40" value=""
           tal:attributes="value context/last" /> </td>
  </tr>
</table>
<input type="submit" name="submit" value="Save Changes" />
</form></div></body></html>
```

ZOPE



BuddyRename in browser.py

```
from zope.event import notify
from zope.app.event.objectevent import ObjectModifiedEvent

class BuddyRename:
    """Rename a buddy"""

    def __init__(self, context, request):
        self.context = context
        self.request = request

    def update(self, first, last):
        self.context.first = first
        self.context.last = last
        notify(ObjectModifiedEvent(self.context))
        self.request.response.redirect("rename.html")
```

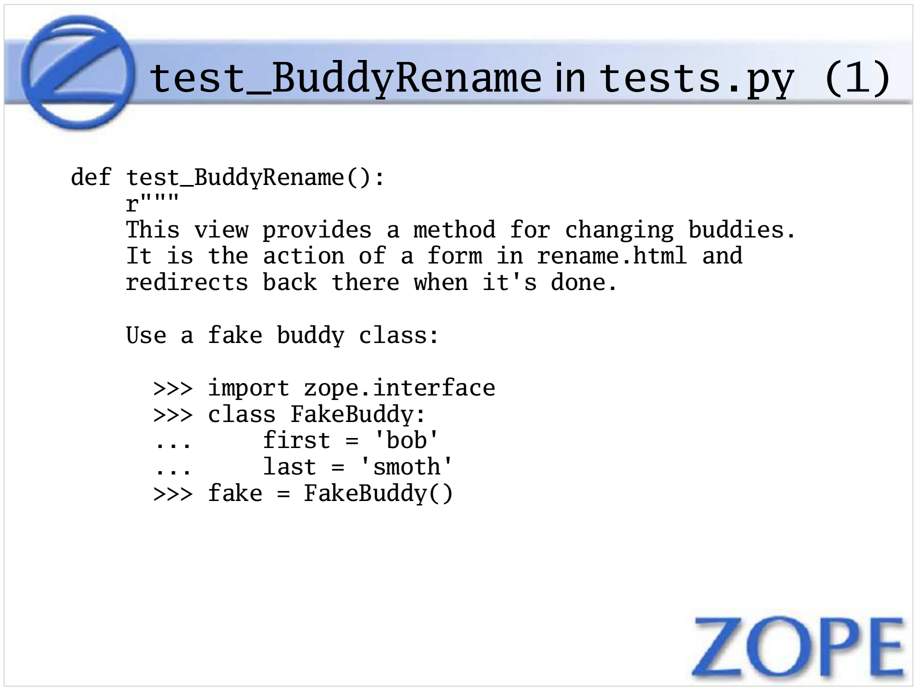
Here we'll use Python code to implement a “page” that serves as the action of a form.

As usual, we have a constructor that takes a context and request and assigns them to attributes.

The update method implements the form action. The Zope publisher will call this method directly, marshaling arguments from form variables. The update method assigns the data passed to it's context's (buddy's) attributes.

The view generates an ObjectModifiedEvent. Events provide a mechanism for plugging logic into **existing** processes. There are a number of activities that we might want to perform, such as updating meta data or catalog indexes when an object is modified. We don't want to make each method that modified an object responsible for these, so, instead we generate an event, and, separately register event subscribers.

Finally, we redirect to the original rename form.

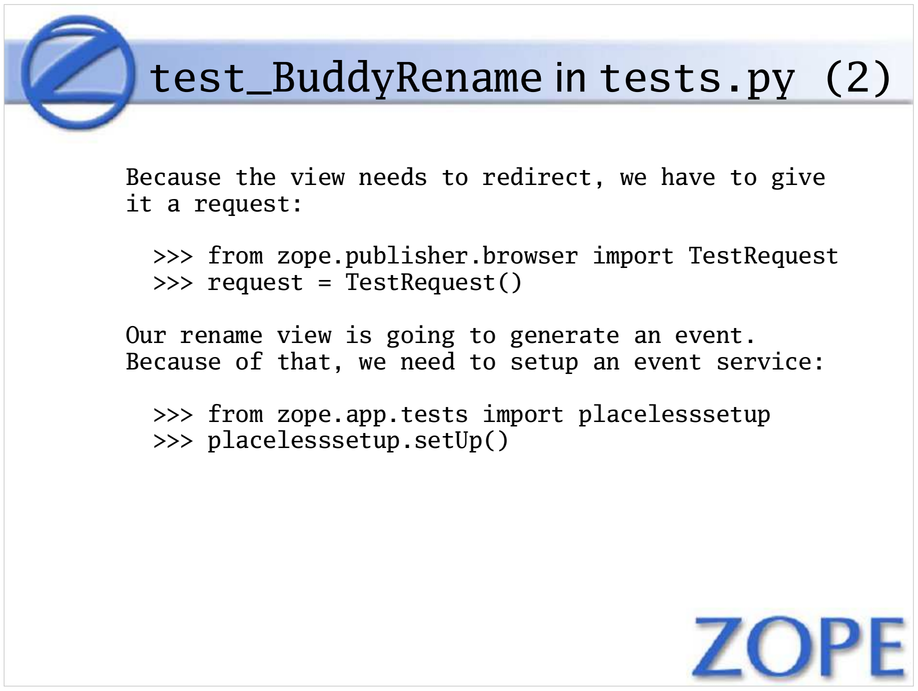


test_BuddyRename in tests.py (1)

```
def test_BuddyRename():
    r"""
    This view provides a method for changing buddies.
    It is the action of a form in rename.html and
    redirects back there when it's done.

    Use a fake buddy class:

      >>> import zope.interface
      >>> class FakeBuddy:
      ...     first = 'bob'
      ...     last = 'smoth'
      >>> fake = FakeBuddy()
```

Again we put the test in the test module. The test code will include a backslash “\”. Whenever we include backslashes in doc tests, we need to mark the doc string as a “raw” string.

Our stub object has some original data that we will change.

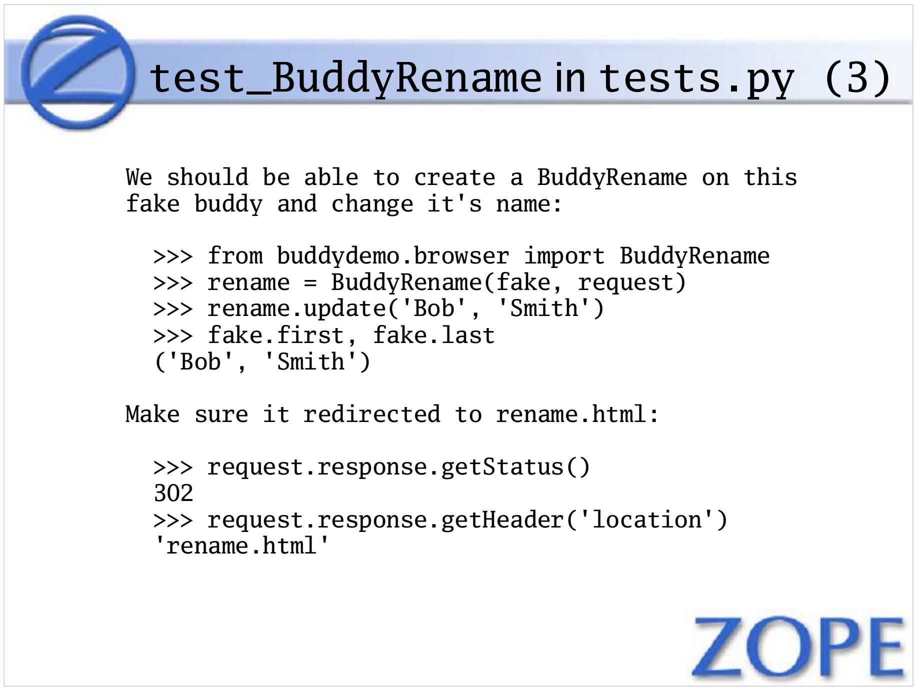


test_BuddyRename in tests.py (2)

```
Because the view needs to redirect, we have to give
it a request:

  >>> from zope.publisher.browser import TestRequest
  >>> request = TestRequest()

Our rename view is going to generate an event.
Because of that, we need to setup an event service:

  >>> from zope.app.tests import placelesssetup
  >>> placelesssetup.setUp()
```

Here we need a real request, because the view is going to use the request response to do a redirect.

Because we publish an event, we need to initialize the event service. `zope.app.tests.placelesssetup.setUp` not only sets up the event service; it also registered a logging event subscriber that we can use to make assertions about generated events.

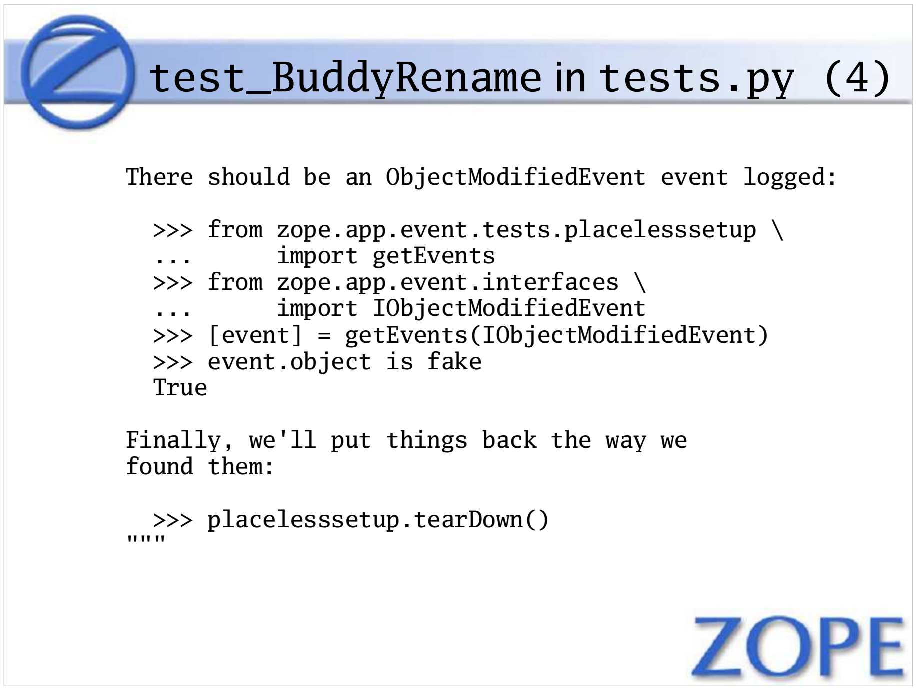


test_BuddyRename in tests.py (3)

```
We should be able to create a BuddyRename on this
fake buddy and change it's name:

  >>> from buddydemo.browser import BuddyRename
  >>> rename = BuddyRename(fake, request)
  >>> rename.update('Bob', 'Smith')
  >>> fake.first, fake.last
  ('Bob', 'Smith')

Make sure it redirected to rename.html:

  >>> request.response.getStatus()
  302
  >>> request.response.getHeader('location')
  'rename.html'
```

We check that calling the update modified the buddy.

We also check to make sure the response has been redirected.

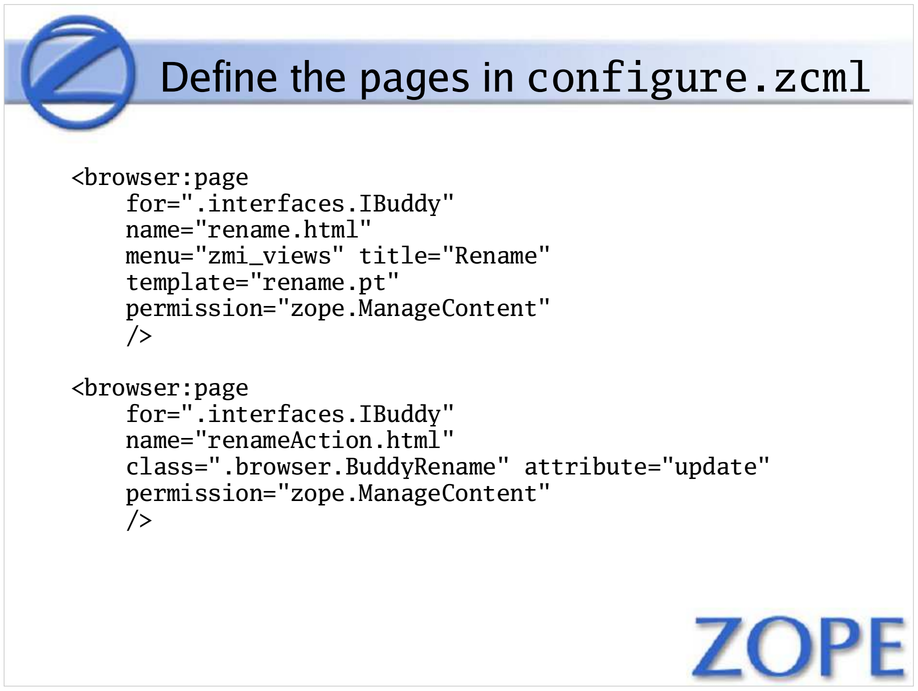


test_BuddyRename in tests.py (4)

```
There should be an ObjectModifiedEvent event logged:

  >>> from zope.app.event.tests.placelesssetup \
  ...      import getEvents
  >>> from zope.app.event.interfaces \
  ...      import IObjectModifiedEvent
  >>> [event] = getEvents(IObjectModifiedEvent)
  >>> event.object is fake
  True

Finally, we'll put things back the way we
found them:

  >>> placelesssetup.tearDown()
"""
```

Finally, we check to make sure that an object-modified event has been generated for our fake buddy.

Notice that we used backslashes to break some imports. This was necessary to fit the source onto a slide for this presentation. The backslashes, in turn, required that we use a raw doc string.

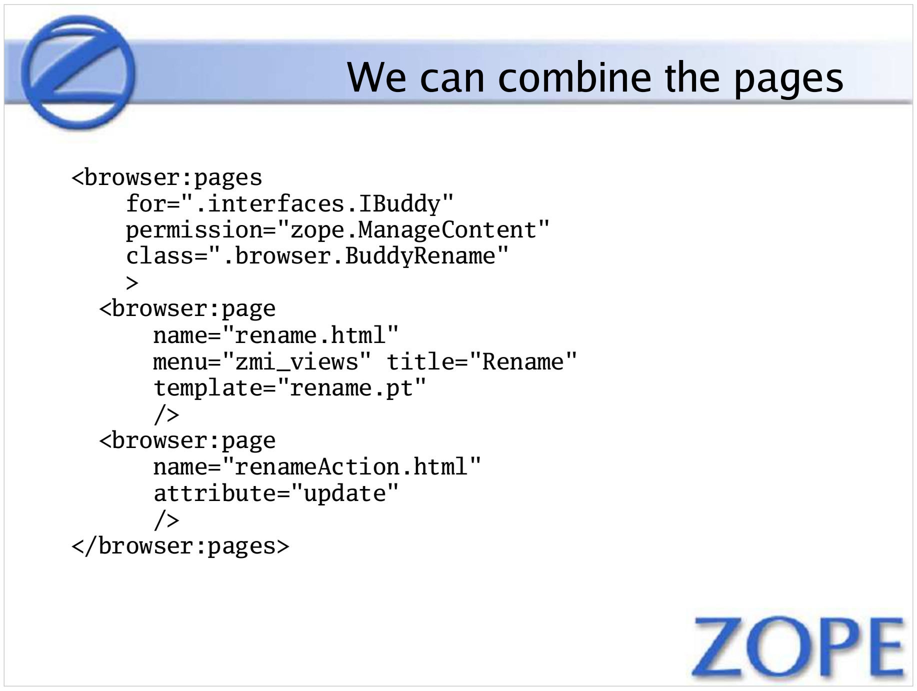


Define the pages in configure.zcml

```
<browser:page
    for=".interfaces.IBuddy"
    name="rename.html"
    menu="zmi_views" title="Rename"
    template="rename.pt"
    permission="zope.ManageContent"
    />

<browser:page
    for=".interfaces.IBuddy"
    name="renameAction.html"
    class=".browser.BuddyRename" attribute="update"
    permission="zope.ManageContent"
    />
```

We've defined two pages. The first page displays the form using the page template, `rename.pt`. As we did for the edit form, we use menu and title attributes to specify a menu item so that we get a “Rename” tab that displays the form.

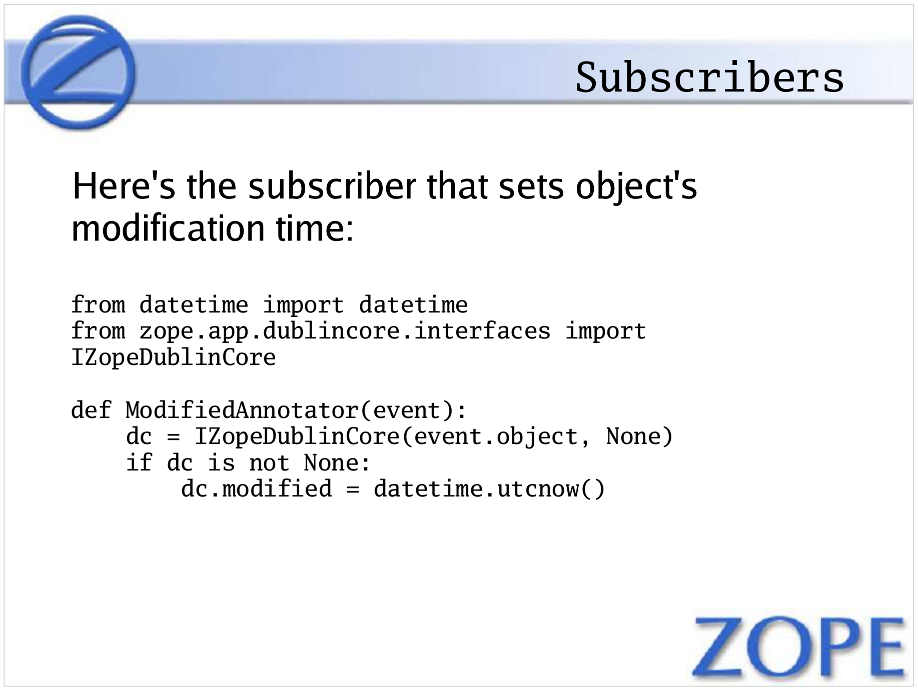
The second page is implemented by a view attribute defined by the class, the update method.



We can combine the pages

```
<browser:pages
    for=".interfaces.IBuddy"
    permission="zope.ManageContent"
    class=".browser.BuddyRename"
    >
  <browser:page
      name="rename.html"
      menu="zmi_views" title="Rename"
      template="rename.pt"
      />
  <browser:page
      name="renameAction.html"
      attribute="update"
      />
</browser:pages>
```

Because the `for` and the `permission` attributes had the same values, we can combine the pages into a `pages` grouping directive. The main benefit of this is to provide some logical grouping.



Subscribers

Here's the subscriber that sets object's modification time:

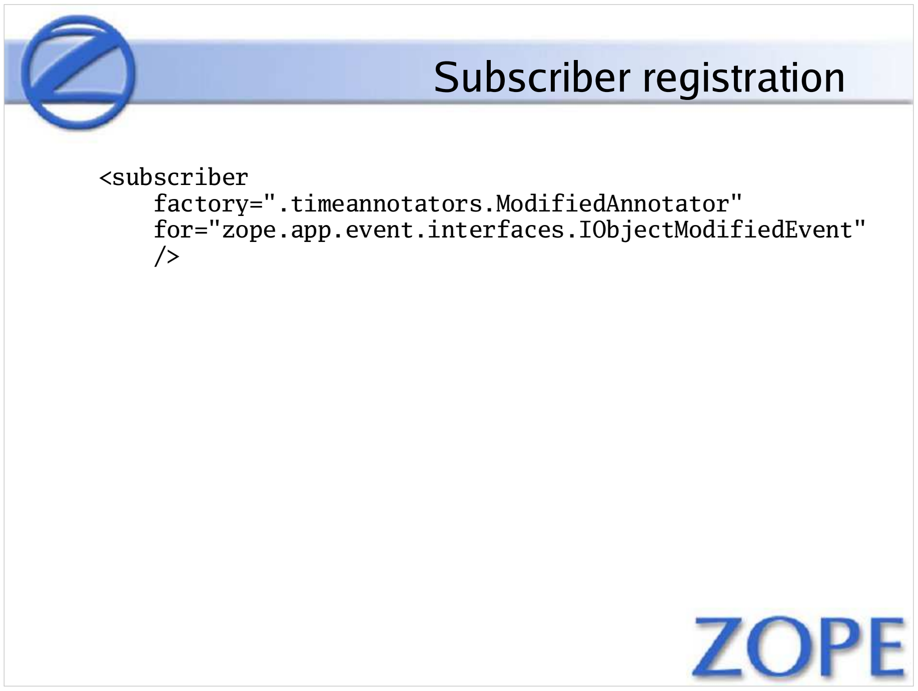
```
from datetime import datetime
from zope.app.dublincore.interfaces import
IZopeDublinCore

def ModifiedAnnotator(event):
    dc = IZopeDublinCore(event.object, None)
    if dc is not None:
        dc.modified = datetime.utcnow()
```

ZOPE

This subscriber is a simple subscriber that takes only an event. Subscribers can be defined to take multiple objects.

Note that **any** object can be an event.

In this example, we used a function rather than a class to define an adapter. Essentially, we're adapting to None. We aren't returning anything useful, but are doing all our work when we are called. This is a compromise of the adapter model, but a justifiable one. Without this compromise, one would generally have to define subscribers with classes or by calling some API function that converted a function to a factory that creates an object that calls the function. It's much cleaner to be able to just use functions as subscribers.

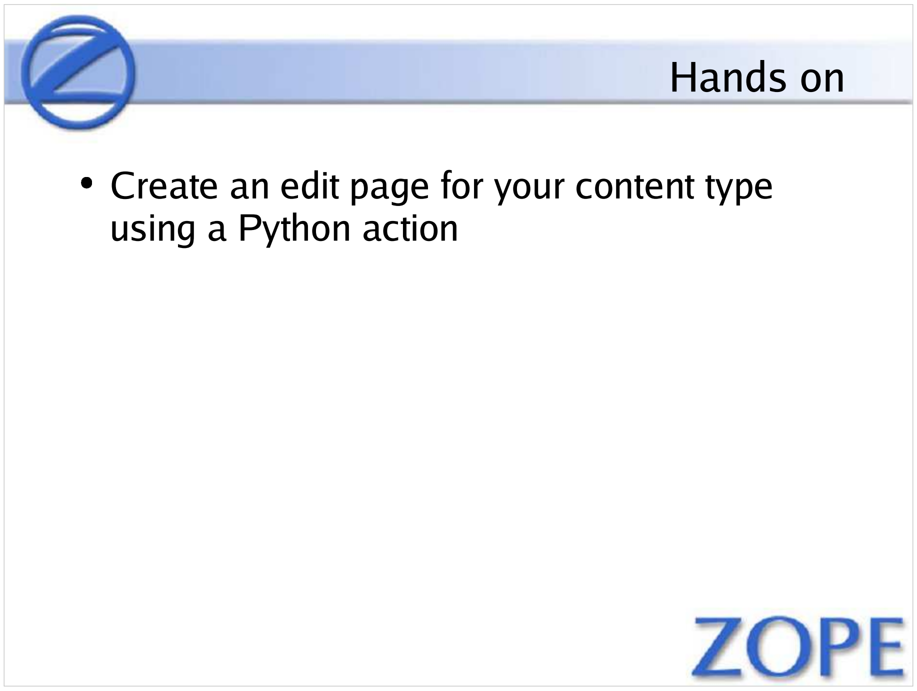


Subscriber registration

```
<subscriber
    factory=".timeannotators.ModifiedAnnotator"
    for="zope.app.event.interfaces.IObjectModifiedEvent"
    />
```

Subscribers can be defined for any number of objects. For subscribers on multiple objects, simply list multiple interfaces in the for attribute, separated by white space.

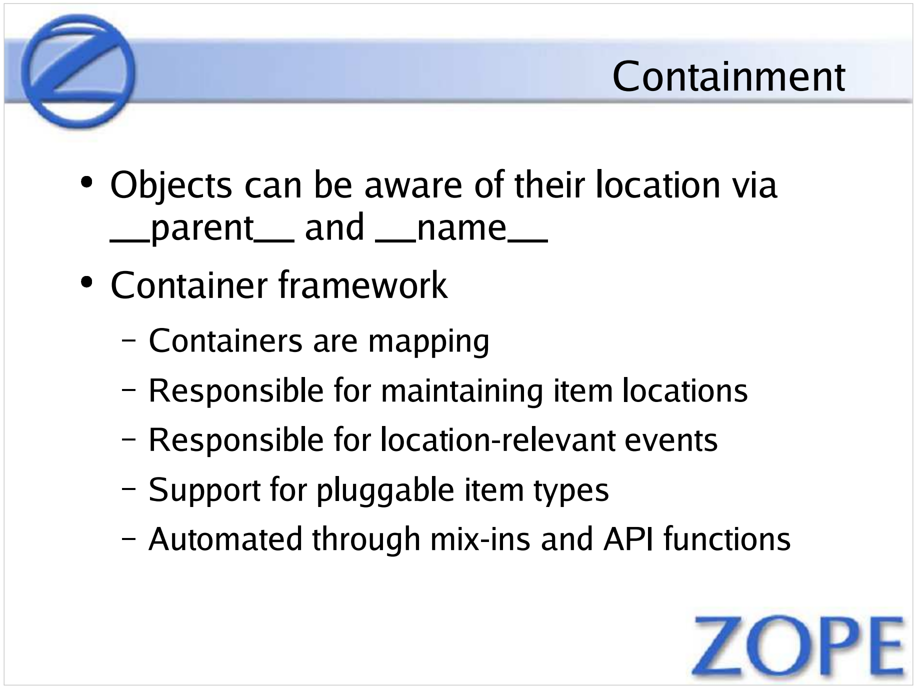
Note that we use a factory attribute here. That's because subscribers are adapters and we specify factories for adapters. This adapter is unusual because we aren't providing an interface. Subscribers can provide an interface, but they don't have to. In fact, most subscribers are just Python functions, as in this example, that do some work when they are called. They are really handlers, not factories. In the future, we'll add a “handler” attribute to this directive to be used when defining handlers rather than factories.



Hands on

- Create an edit page for your content type using a Python action

ZOPE



Containment

- Objects can be aware of their location via __parent__ and __name__
- Container framework
  - Containers are mapping
  - Responsible for maintaining item locations
  - Responsible for location-relevant events
  - Support for pluggable item types
  - Automated through mix-ins and API functions

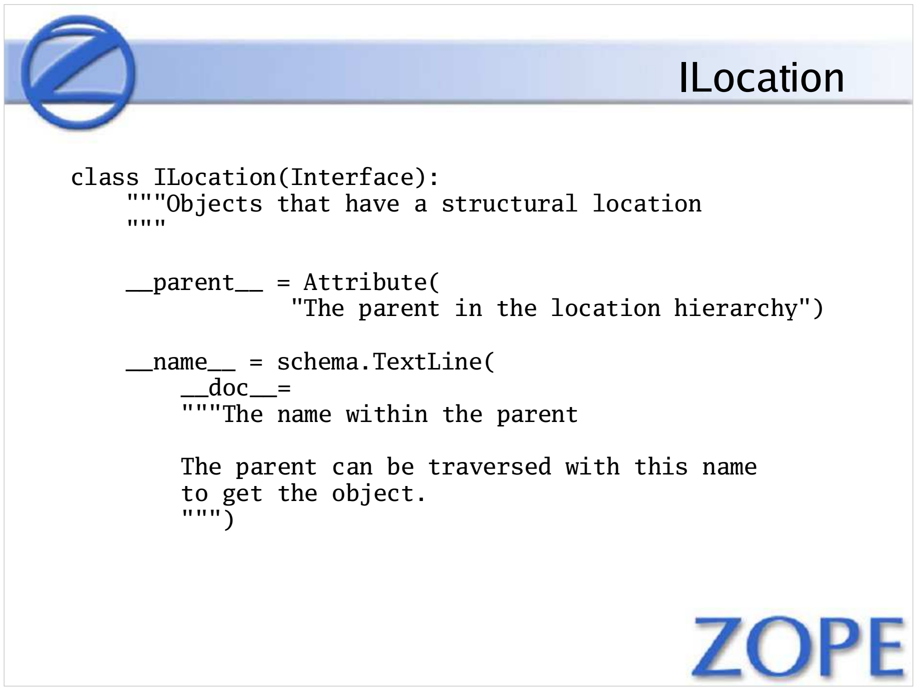
ZOPE

The main purpose of the container framework is to create containers that can hold many different kinds of objects, including objects not created by the container authors. The framework provides mechanisms to decide which kinds of objects a container can hold and which containers an object can be placed in. If you don't need this flexibility, then you are free not to use the framework.

In addition to providing a mapping protocol, containers are responsible for making sure that their items have location information. This may require placing containment proxies around items to assure that they implement ILocation and setting the item's `__parent__` and `__name__` attributes.

In addition, when containers are modified, they need to generate location-relevant events.

Carrying out these responsibilities is quite involved. Fortunately (or unfortunately, depending on your point of view), there are some base classes and utility functions that automate these responsibilities.

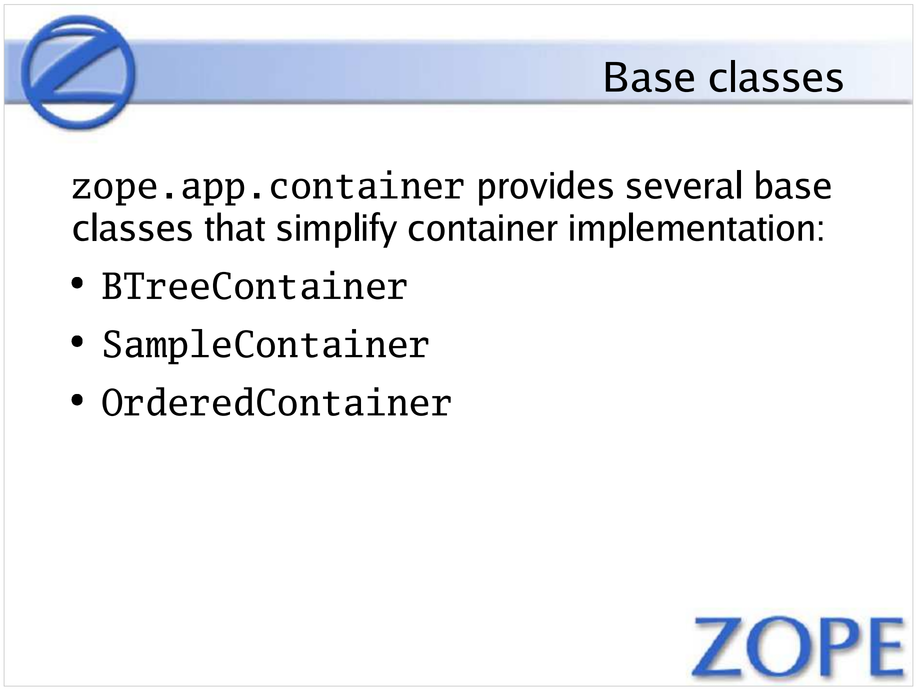


ILocation

```
class ILocation(Interface):
    """Objects that have a structural location
    """

    __parent__ = Attribute(
                "The parent in the location hierarchy")

    __name__ = schema.TextLine(
        __doc__=
        """The name within the parent

        The parent can be traversed with this name
        to get the object.
        """)
```

`ILocation` specifies basic location information. It allows us to perform acquisition and to compute object locations.

In Zope 3, we store location information directly, rather than through transient context wrappers. An object has a single canonical location. An object can have many references, and, from a reference, you can compute an object's true location.



Base classes

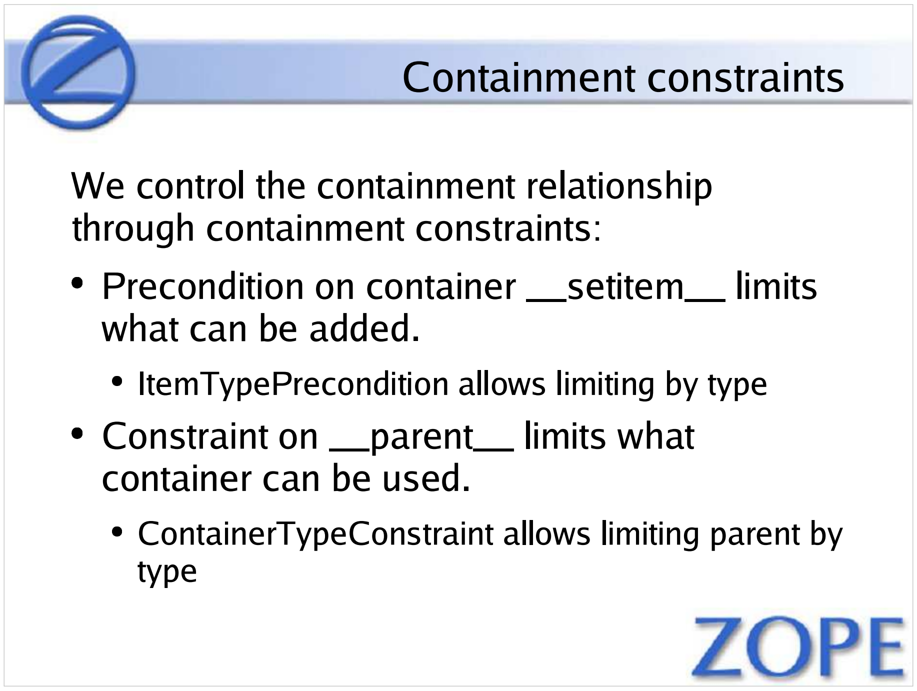
`zope.app.container` provides several base classes that simplify container implementation:

- `BTreeContainer`
- `SampleContainer`
- `OrderedContainer`

BTreeContainer is the most commonly used. It supports very large containers.

SampleContainer is rarely used. It provides a hook for specifying lower-level storage. BTreeContainer subclasses this.

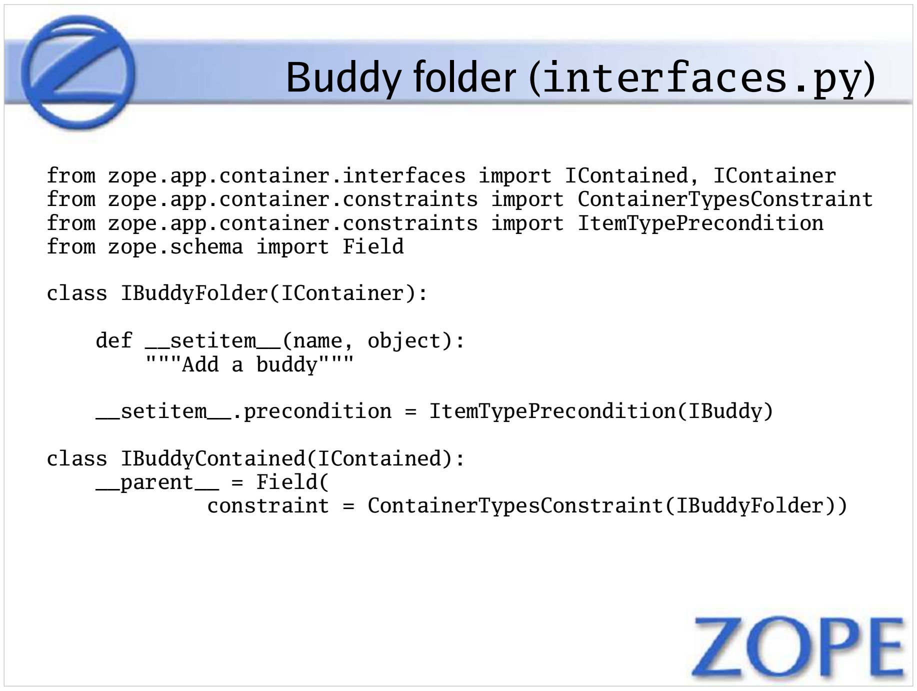
OrderedContainer provides for ordered items. An API is provided for manipulating order. This should not be used for larger containers.



Containment constraints

We control the containment relationship through containment constraints:

- Precondition on container __setitem__ limits what can be added.
  - ItemTypePrecondition allows limiting by type
- Constraint on __parent__ limits what container can be used.
  - ContainerTypeConstraint allows limiting parent by type



Buddy folder (interfaces.py)

```
from zope.app.container.interfaces import IContained, IContainer
from zope.app.container.constraints import ContainerTypesConstraint
from zope.app.container.constraints import ItemTypePrecondition
from zope.schema import Field

class IBuddyFolder(IContainer):

    def __setitem__(name, object):
        """Add a buddy"""

    __setitem__.precondition = ItemTypePrecondition(IBuddy)

class IBuddyContained(IContained):
    __parent__ = Field(
             constraint = ContainerTypesConstraint(IBuddyFolder))
```

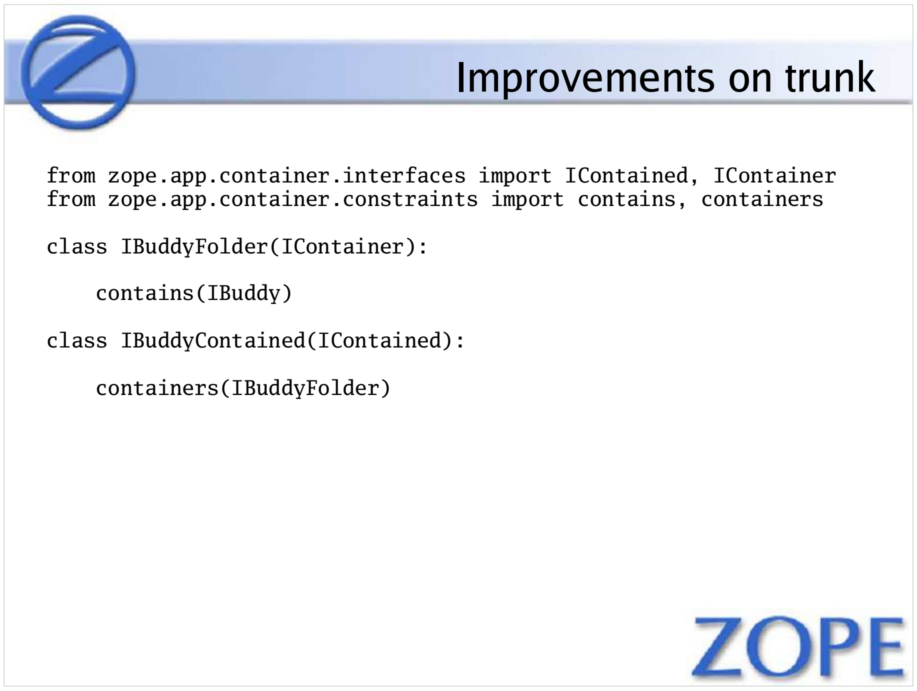
ZOPE

Preconditions are tagged values on interface attribute definitions. When defining interfaces with class statements, we express the preconditions as function attributes. We can also express preconditions after an interface has been created. To do so, we use the getitem operation on the interface to get an attribute definition and then use the `setTaggedValue` method on the definition to set the value:

```
IContactFolder['__setitem__'].setTaggedValue(
    precondition', ItemTypePrecondition(IContact)
```

We decided not to modify `IBuddy`. There are a number of reasons why we did this:

- We didn't want `__parent__` to become part of `IBuddy's` schema. This would have complicated form generation.
- We didn't want to require all `IBuddys` to be contained
- By creating a separate interface, we avoided a circular dependency between `IBuddyFolder` and `IBuddyConstrained`.

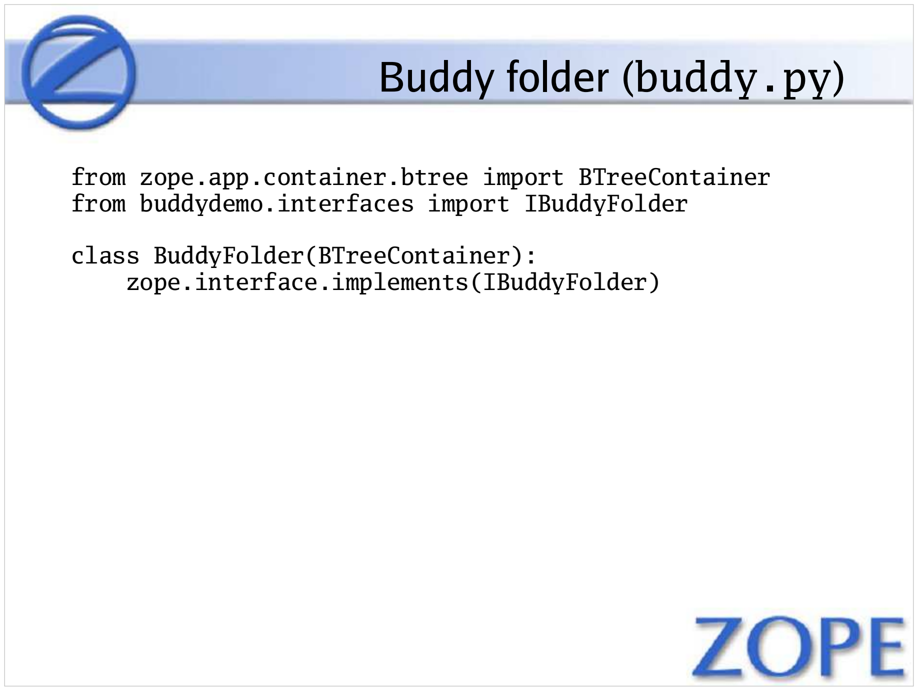


Improvements on trunk

```
from zope.app.container.interfaces import IContained, IContainer
from zope.app.container.constraints import contains, containers

class IBuddyFolder(IContainer):

    contains(IBuddy)

class IBuddyContained(IContained):

    containers(IBuddyFolder)
```

On the subversion trunk (and in the next release, Zope X3.1) there are improved APIs for defining containment constraints. The lower-level mechanisms defined in the previous slide are still supported, and are necessary in some cases, however, they are rather error prone.

One problem with this mechanism is that what we often really want to do is to constrain a relationship between two types. This is a bit clumsy to express as properties of the individual types. It makes more sense in some ways to express the constraint independently of either type, perhaps as some sort of subscriber.

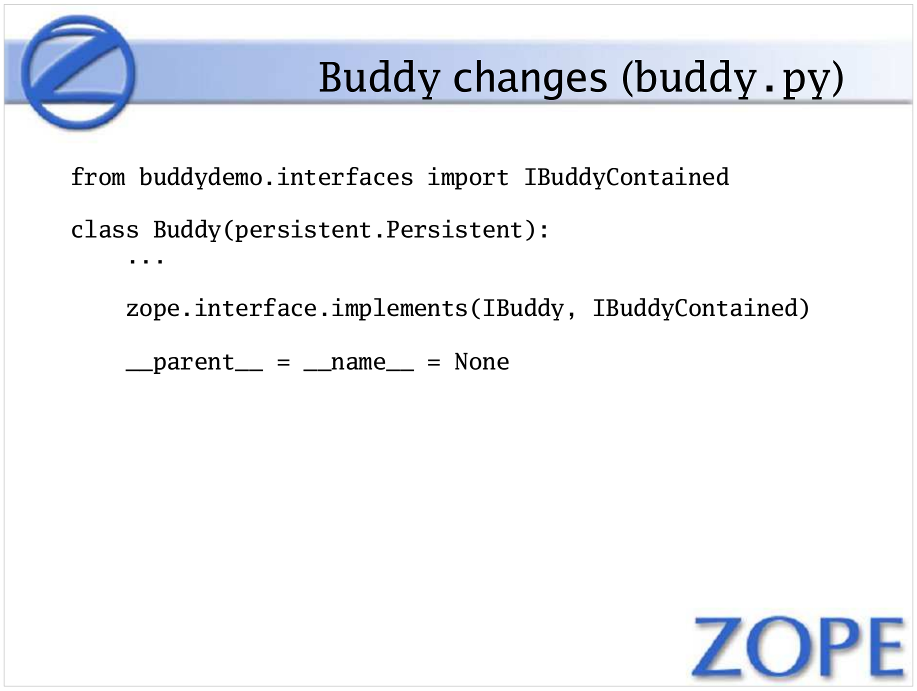


Buddy folder (buddy.py)

```
from zope.app.container.btree import BTreeContainer
from buddydemo.interfaces import IBuddyFolder

class BuddyFolder(BTreeContainer):
    zope.interface.implements(IBuddyFolder)
```

Here, all we do is subclass `BTreeContainer` and add an interface.

Alternatively, we could provide a factory that simply instantiated `BTreeContainer`s and provided instance-specific interface declarations.

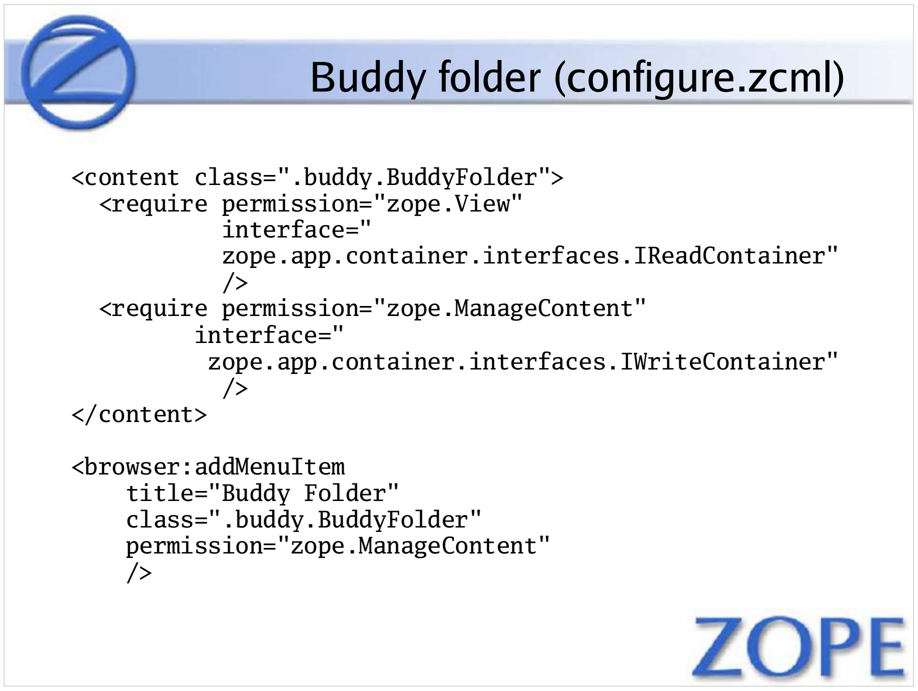


Buddy changes (buddy.py)

```
from buddydemo.interfaces import IBuddyContained

class Buddy(persistent.Persistent):
    ...

    zope.interface.implements(IBuddy, IBuddyContained)

    __parent__ = __name__ = None
```

Here we added an extra interface to the declaration for the buddy class.

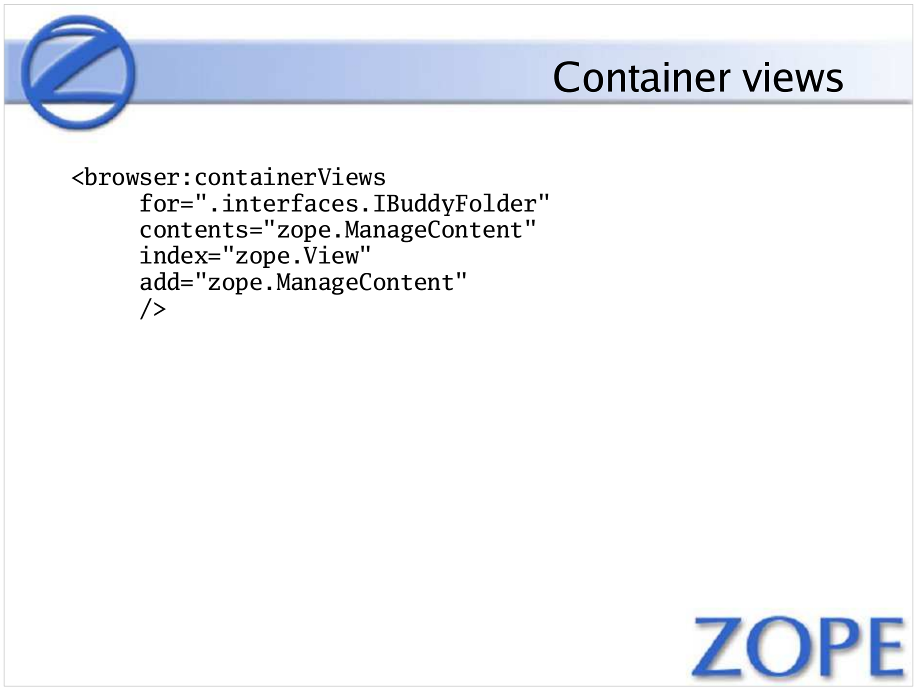
Because `IContained` requires `__parent__` and `__name__` attributes, we need to provide default values for them. It's easiest to do so by providing default values at the class level. Alternatively, we could have modified the `__init__` method. We could also have subclasses zope.app.container.Contained, which provides this trivial implementation.



Buddy folder (configure.zcml)

```
<content class=".buddy.BuddyFolder">
  <require permission="zope.View"
           interface="
           zope.app.container.interfaces.IReadContainer"
           />
  <require permission="zope.ManageContent"
         interface="
          zope.app.container.interfaces.IWriteContainer"
           />
</content>

<browser:addMenuItem
    title="Buddy Folder"
    class=".buddy.BuddyFolder"
    permission="zope.ManageContent"
    />
```

Containers provide read and write interfaces, which we need to make security declarations for. (This borders on being a dead chicken.)

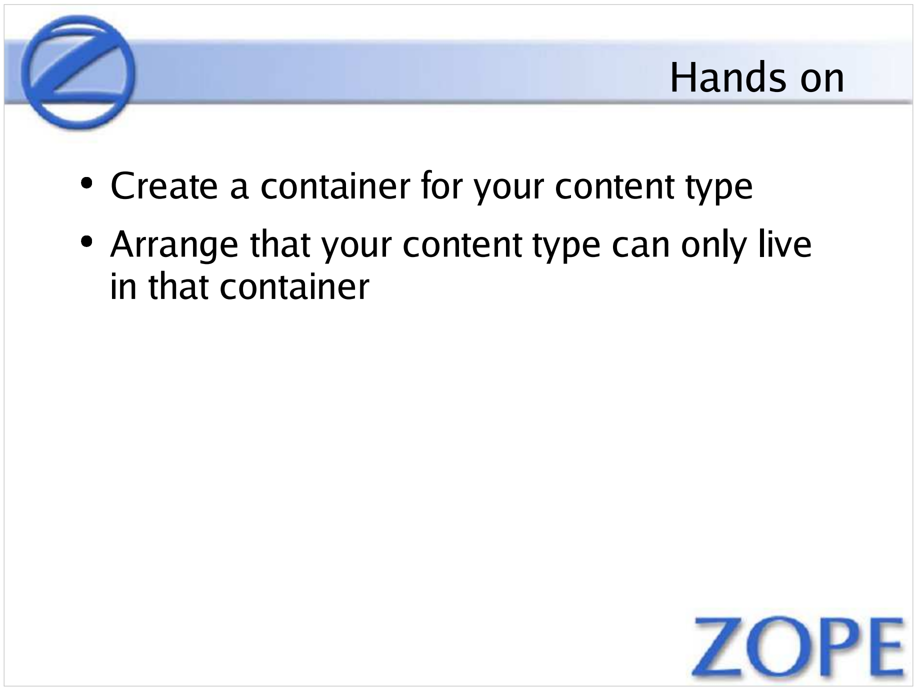


Container views

```
<browser:containerViews
     for=".interfaces.IBuddyFolder"
     contents="zope.ManageContent"
     index="zope.View"
     add="zope.ManageContent"
     />
```

Specify one or more of contents, index, or add attributes. For each of these attributes, a view will be defined requiring the permission given as an attribute value.

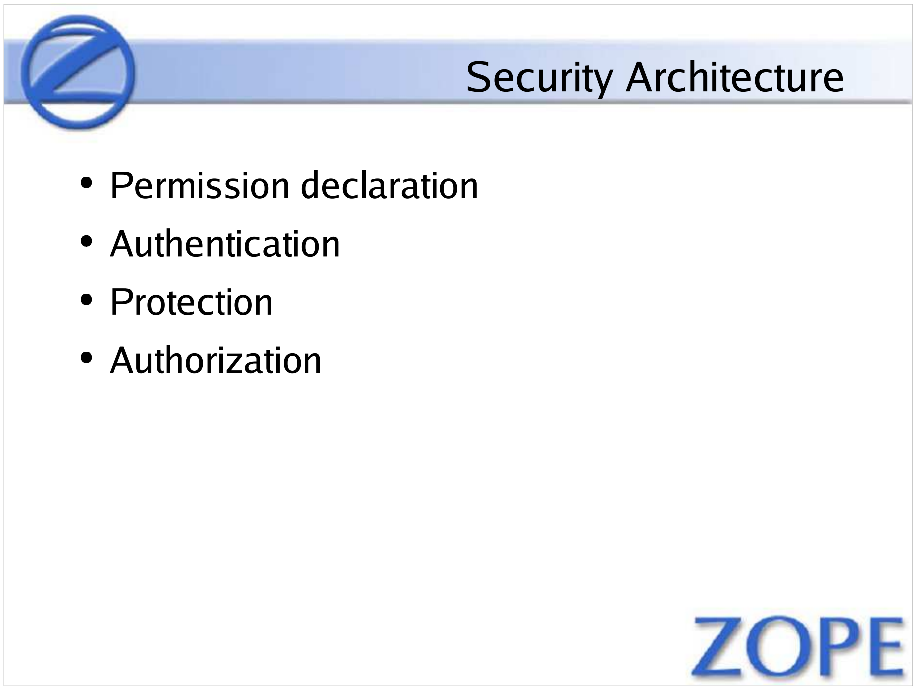
This directive is, effectively, a macro. It generates (actually calls the ZCML handlers for) the more detailed directives on the previous slide.



Hands on

- Create a container for your content type
- Arrange that your content type can only live in that container

ZOPE



Security Architecture

- Permission declaration
- Authentication
- Protection
- Authorization

ZOPE

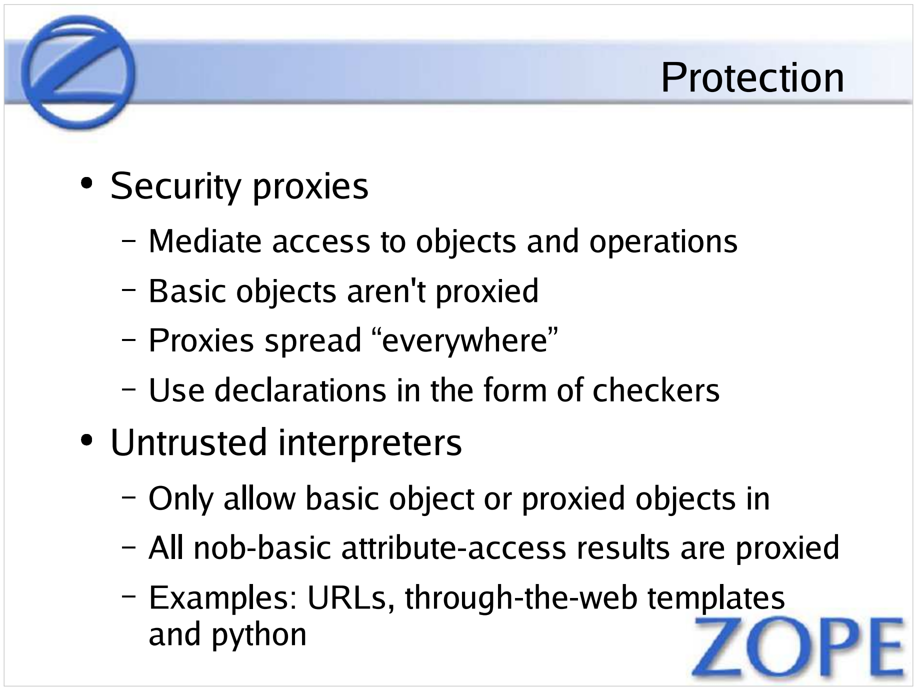
We use ZCML to declare permissions needed to access names in classes or to use certain components.

We use authentication services to extract credentials from a request and give us principals. Principals are entities that we can grant access to. (The details of what kind of grants we can make and how are determined by the authorization system.

The protection system is responsible for enforcing security in a Zope application. The protections system prevents access to attributes or operations unless “interactions” have required permissions. The protection system uses the authorization system to determine whether “interactions” have the needed permissions.

An interaction is the use of one or more external entities with the system. A common case is that a user interacts with the system by making a web request. The authentication system is used to determine a principal corresponding to the user and the principal is associated with the interaction through the request. Generally, an interaction has a permission on an object if each of it's principals do.

The authorization system is pluggable. It is responsible for making authorization decisions and for managing the grants used to make those decisions. The authorization system determines what kinds of grants can be made and provides mechanisms for making and managing the grants.



Protection

- Security proxies
  - Mediate access to objects and operations
  - Basic objects aren't proxied
  - Proxies spread “everywhere”
  - Use declarations in the form of checkers
- Untrusted interpreters
  - Only allow basic object or proxied objects in
  - All nob-basic attribute-access results are proxied
  - Examples: URLs, through-the-web templates and python

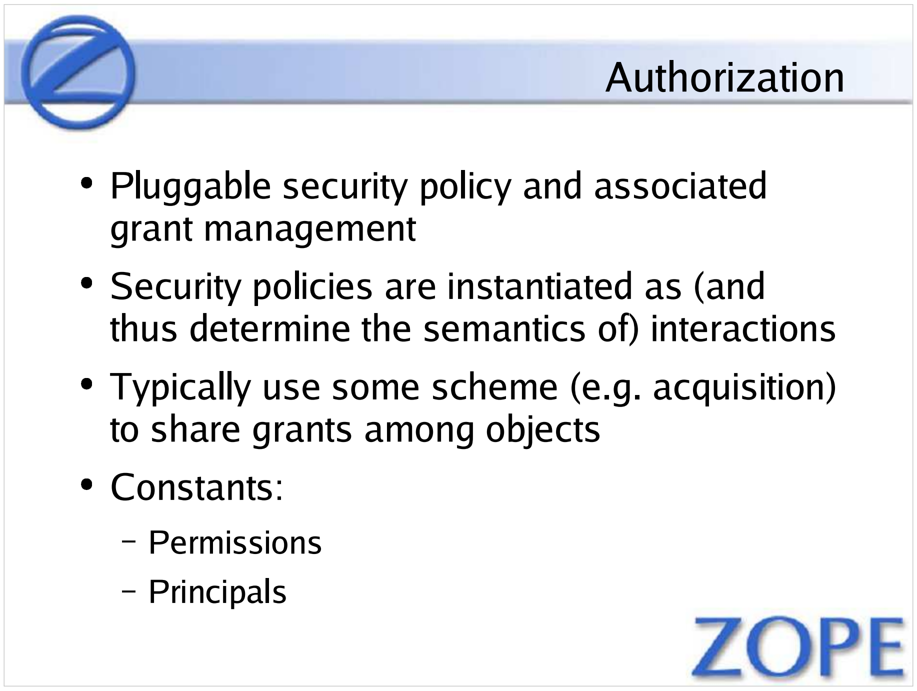
The protection system makes sure that interactions have the required permissions.

The central protection mechanism is security proxied. When an object enters untrusted code (e.g. when traversing a URL):

- We pass the object to a proxy factory
- We look up a checker for the object. We may get a result indicating that the object is “basic”, meaning we don't need to proxy it.
- Otherwise, we get a checker object. We create a proxy around the object, passing the checker. The proxy delegates to the checker to:
  - Make access decisions
  - Create new proxies

Basic objects are immutable and contain immutable data. Examples are objects like strings, numbers, and dates (but not tuples).

Checkers represent permission declarations. They are looked up for individual instances, or for their classes. They may be found either as __Security_checker__ attributes or in a special registry.



Authorization

- Pluggable security policy and associated grant management
- Security policies are instantiated as (and thus determine the semantics of) interactions
- Typically use some scheme (e.g. acquisition) to share grants among objects
- Constants:
  - Permissions
  - Principals

ZOPE

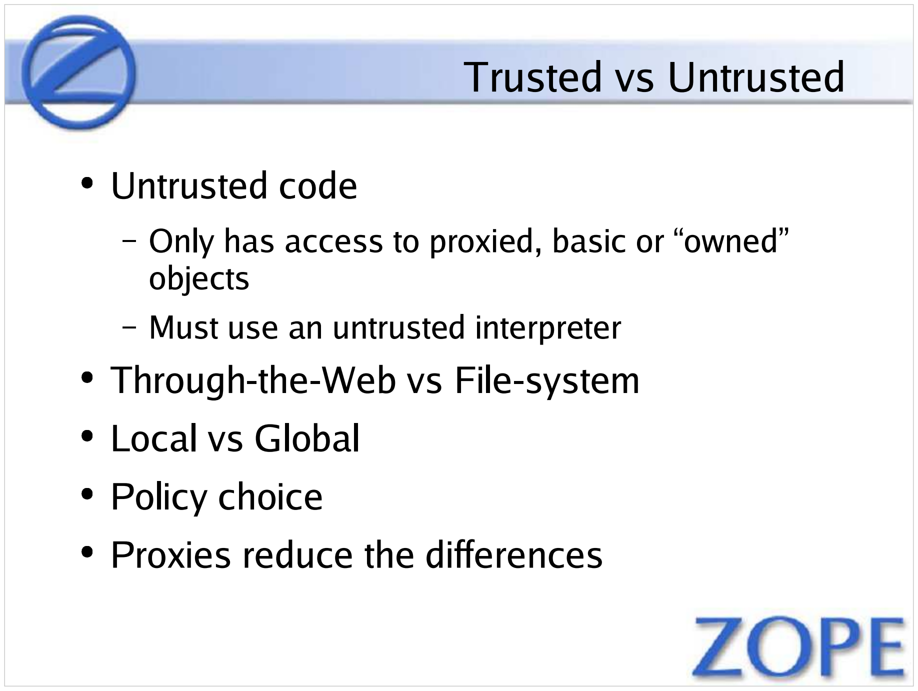
Typically, the security policy is simple an interaction class. Interactions are stored as thread-local data. The protections system gets an interaction for the current thread and calls a method on it to determine if it has a permission **on an object.**

Currently have a “classic” role-based security policy:

- principal roles
- role permissions
- principal permissions
- allow and deny
- acquired grants

In this example, the use of roles is **specific** to this security policy, as is the use of acquisition.

We're planning a major change in the basic model by allowing hierarchical principals (groups) and permissions (permission sets). This largely eliminates the need for roles.



Trusted vs Untrusted

- Untrusted code
  - Only has access to proxied, basic or “owned” objects
  - Must use an untrusted interpreter
- Through-the-Web vs File-system
- Local vs Global
- Policy choice
- Proxies reduce the differences

ZOPE

See zope/security/untrustedinterpreter.txt for a detailed definition of an untrusted interpreter.

Usually (e.g. X3.0) through-the-web (TTW) code is untrusted and file-system based code is trusted.

Usually, local components are TTW, but this need not be the case.

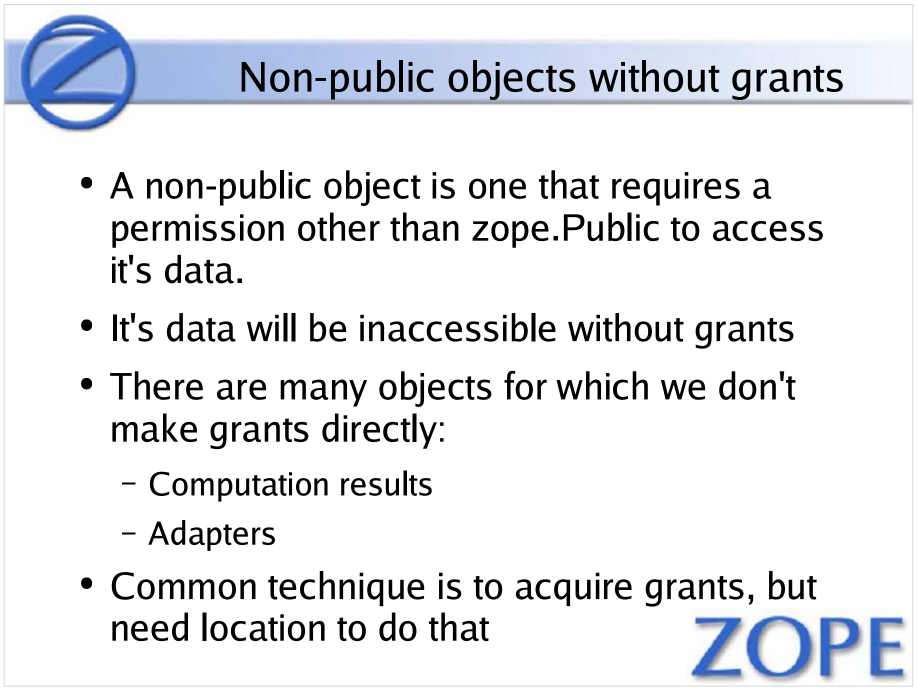
Global components are always file-system based.

Someday, it will be possible to set policy for when code is trusted. Sites will be able to say that local or TTW code is trusted.

Because of the way proxies multiply (operations on proxies return proxies), even trusted code is mediated by the security system. This is a good thing because:

– It makes it much harder to trick trusted code into performing operations inaccessible to untrusted code

– It makes it harder to ignore security



Non-public objects without grants

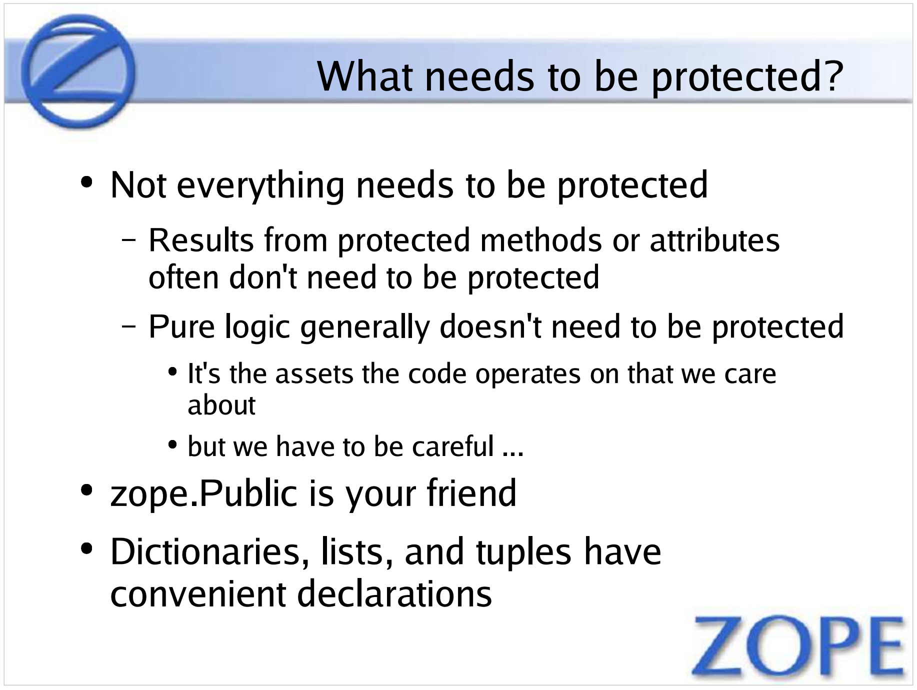
- A non-public object is one that requires a permission other than zope.Public to access it's data.
- It's data will be inaccessible without grants
- There are many objects for which we don't make grants directly:
  - Computation results
  - Adapters
- Common technique is to acquire grants, but need location to do that

I suspect that this is the thorniest aspect of Zope 3 development. That's not surprising, as it's probably the thorniest aspect of Zope 2 development too.

It's “worse” for Zope 3 because the protection system is more robust. Even trusted code is often subject to the security system by virtue of being passed untrusted code.

Functional tests are very helpful for detecting security problems.

Key issue is objects without grants. They are much easier to create that you might expect. This most commonly applies to transient objects. Even if you make grants on **all** of your content objects, you still have to content with the protection of computation results.



What needs to be protected?

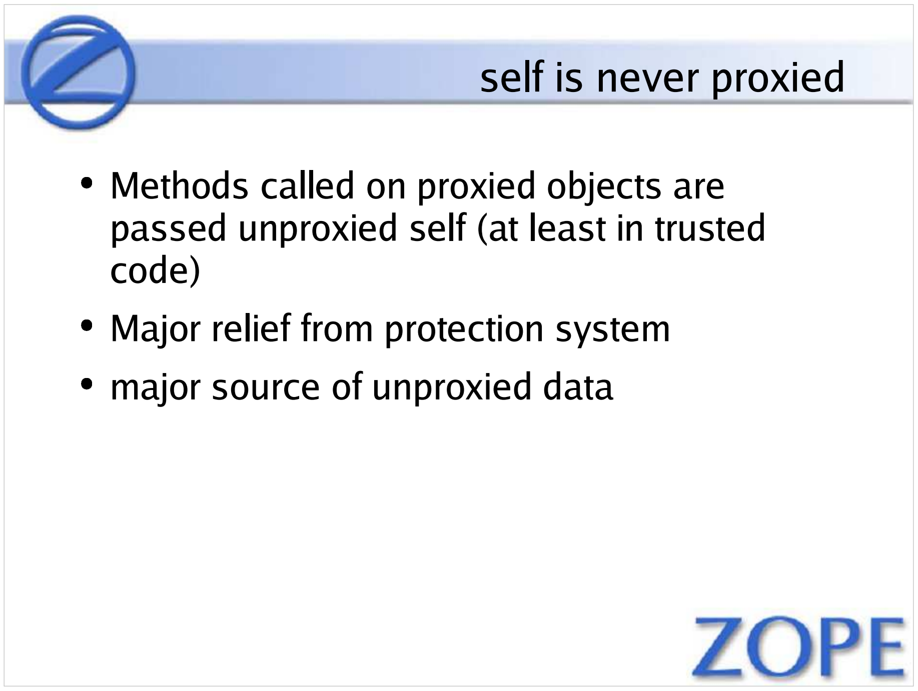
- Not everything needs to be protected
  - Results from protected methods or attributes often don't need to be protected
  - Pure logic generally doesn't need to be protected
    - It's the assets the code operates on that we care about
    - but we have to be careful ...
- zope.Public is your friend
- Dictionaries, lists, and tuples have convenient declarations

If a permission is needed to call a method, then you might not need to protect the result. Protecting the method effectively prevents unauthorized access to the result. If the result is a basic value or a dictionary, tuple, or list, then we don't need to do much else.

If a result is an application-defined instance, then you will need to make security declarations. Consider using zope.Public.

It is always “safe” to return dictionaries and lists. The declarations of these allow unfettered access to their data, but prevent modification.

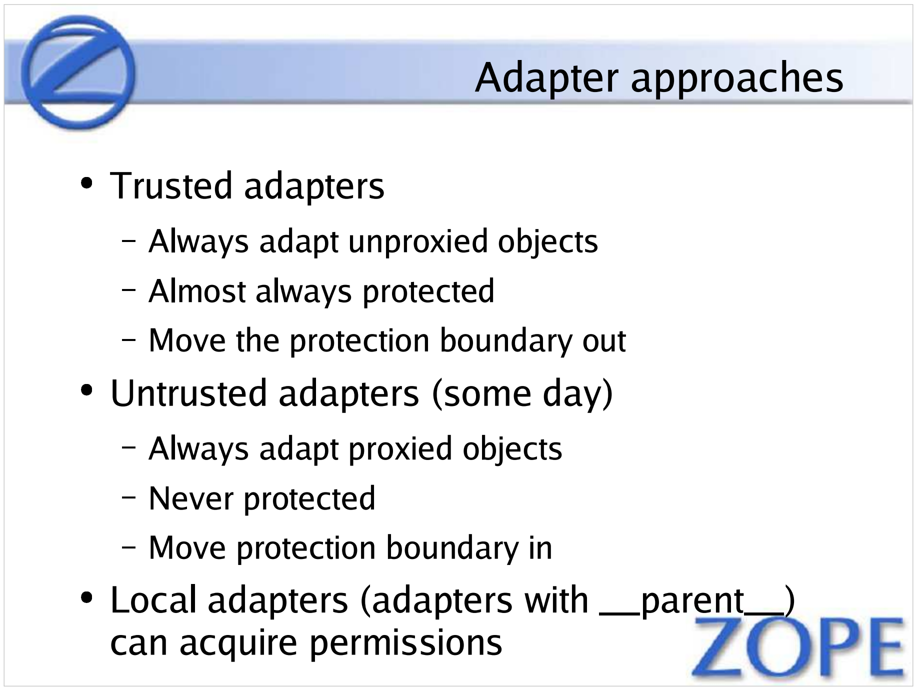
In theory, we almost never need to protect code. It's not the code we care about, but the objects it accesses. In particular, if adapters adapted security proxies, there would be no point in preventing access to the adapters, since protection is provided by the proxies around the adapted objects. Unfortunately, there's no way to guarantee this at this time.



self is never proxied

- Methods called on proxied objects are passed unproxied self (at least in trusted code)
- Major relief from protection system
- major source of unproxied data

ZOPE



Adapter approaches

- Trusted adapters
  - Always adapt unproxied objects
  - Almost always protected
  - Move the protection boundary out
- Untrusted adapters (some day)
  - Always adapt proxied objects
  - Never protected
  - Move protection boundary in
- Local adapters (adapters with __parent__) can acquire permissions

ZOPE

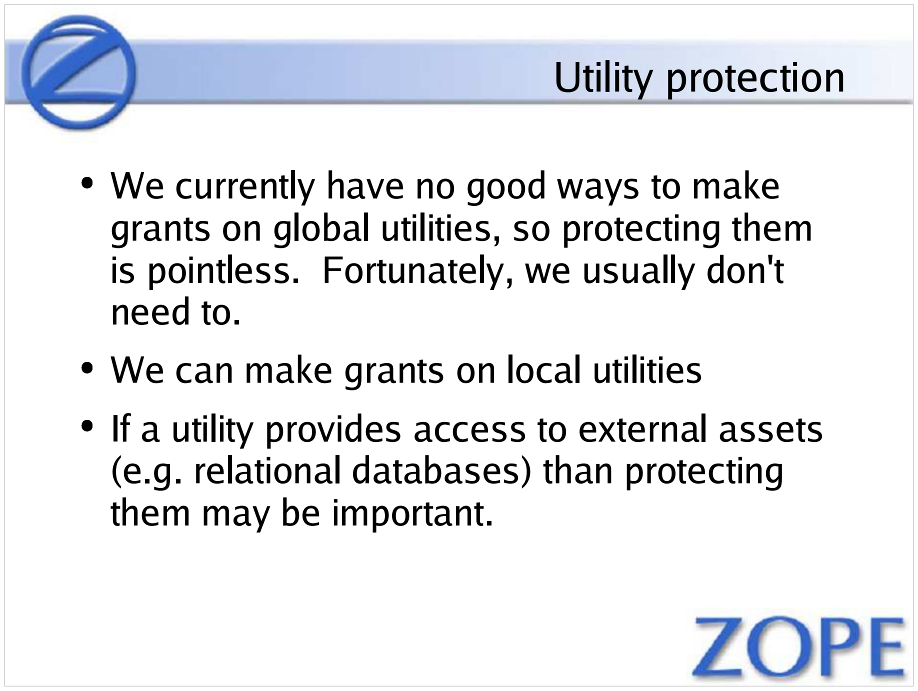
Trusted adapters are very useful when you have APIs that you never want to expose directly to untrusted code. You can provide indirect access through trusted adapters. When a trusted adapter is created for a proxied object, the proxy is removed and a new proxy is created around the resulting adapter.

Untrusted adapters will allow us to avoid protecting code. Rather than protecting the code, we'll rely on the protections on the adapted objects. Most adapters behave this way today as a consequence of the way they are usually accessed. For example, views are almost always created from untrusted code (URLs). Adapters created from trusted code may not adapt proxies, if the trusted code has access to unproxied data.

The thing that makes untrusted adapters difficult to implement is that we don't want to create security proxies unless we need to. This often means that we can't create the proxies until after an adapter has been created. To make this work, we'll need to invent some sort of "re-adaptation" mechanism.

A simple, if intrusive technique is to provide adapters with __parent__ attributes. This is often set to the adapter context, so that an adapter can acquire grants from it's context. Views created with the ZCML browser directives implement Ilocation and get __parent__ set automatically.

The good news is that adapters are usually not used from untrusted code, so their protection is less critical.



Utility protection

- We currently have no good ways to make grants on global utilities, so protecting them is pointless. Fortunately, we usually don't need to.
- We can make grants on local utilities
- If a utility provides access to external assets (e.g. relational databases) than protecting them may be important.

ZOPE

Utilities are generally used from trusted code, so protecting them is not crucial.

Utilities that encapsulate algorithms are not a problem, as they can be declared to require zope.Public. We don't need to protect code.

Utilities that provide access to external assets are more problematic. The simplest approach is to make them local and store them in site-management folders, where we can make grants for them.