vinstall Documentation

Release a0.1

rbistolfi

Jul 15, 2017

Contents

1 Overview **3**
1.1 The Application object 4
1.2 Model objects 4
1.3 View objects 5
1.4 Registering new views 5

Vinstall is an application toolkit for the Vector installer, implementing a MVCish framework.

CHAPTER 1

Overview

An application written using **vinstall** usually consists in just a set of controller classes, implementing a required **interface**. Each controller class represent a state in the application and they have the following responsibilities:

- Defining the next controller class
- Defining the previous controller class
- Defining the information that will be rendered in the screen
- Reacting to user input

The first two are implemented by defining a `next()` and a `previous()` methods, returning the classes representing the next and the previous state of the application. They usually will contain some simple logic, because sometimes the next step will depend on the state of the application or the environment. In the same way, returning to a previous state of the application could require some cleaning up from your side. The `render()` method is used to present information to the user. It should return a `Render` instance. A `Render` object is created with at least three arguments. The first is a `str()` object used for the title of this stage of your app. The second one is also a `str()` object, representing a introductory text that will be shown right next to the title. Finally, one or more **model** objects. Model objects are not very special. The only property they have is that a **view** has been registered for them. We provide a set of model objects representing common form elements in the *vinstall.core.model* module. You can also create your own models and register views for them (more on this later). Finally, controllers can react to user input in two ways, by defining a `command()` method and/or a `process()` method. Both methods take as many arguments as model objects you passed with the `render()` method. The `process()` method is called inmediatelly after user requests the next state (typically when she clicks "next") and before the next stage is shown. The `command()` method is scheduled for later execution, and it will be called after all the controller classes have been processed. So, a controller class looks like this:

```python
from vinstall.core.render import Render
import vinstall.core.model as model

class MyController(object):
    """A controller

    """
    def render(self):
        """A method returning a Render instance.
        The first to args of the Render's constructor are a title
        and an intro text. The rest of the arguments are model objects
        The BooleanOption will be rendered as a checkbox

        """
        return Render("Hello world", "This is the intro",
               model.BooleanOption("This is a boolean option")

    def next(self):
        """Return the next controller class. If this returns None, we
        assume it is the end of the application.

        """
        return None

    def previous(self):
        """Return the previous controller class.

        """
        return TheFirstController

    def command(self, boolean):
        """The signature matches the number of model objects in the
        render method. This will be executed later.

        """
        if boolean:
           myapp.do_something()
```

The Application object

Finally, you just need to start you app by creating an `Application` instance, passing the first controller class (first as in the one representing the initial state of your app) and the name of a view module as a string. There are two views defined for the provided model objects, “urwid” and “gtk”. You can start the app using the `run()` method:

```
app = Application(MyFirstController, "urwid")
app.run()
```

This should be all you need to know for writting a simple app. Below there is a small overview of the other objects used in the application, so you can understand better whats going on.

Model objects

Model objects do not need special behavior. Usually, it will be just an object from your domain, or business layer. Ocasionally, your model objects will need to implement special behavior, such as *observable* objects or *persistent* objects. In general, the model type is only important for finding a registered view for it, so you don’t really need to worry about it.

View objects

View objects are the graphical representation of your model objects. We provide a simple class decorator for registering views against model types. *View* objects need to implement a required **Interface** defined in the *core.view.View* class. This class is only a reference and you don't need to subclass it for creating a View type.

Registering new views

New views can be registered using the `renders()` class decorator:

```
@renders(my.model.Class)
class MyModelClassView(object):
    """A view for MyModelClass

    """
```

For more information, check out the module index.

- genindex
- modindex
- search