Ajenti

Release 1.2.22.0

June 08, 2015

CHAPTER 1

Documentation

1.1 FAQ

1.1.1 How do I add domains/PHP/email accounts/websites?

Pure Ajenti is a **server control** panel, not a **hosting control** panel. You need the Ajenti V add-on for web-hosting stuff: http://ajenti.org/#product-ajenti-v.

1.1.2 I forgot my password

Open /etc/ajenti/config.json, look for your user entry, and replace whole password hash entry with a new plaintext password. Restart Ajenti. Click "save" under "Configuration" to rehash the password.

1.1.3 My OS isn't supported, but I'm a brave adventurer

```
pip install ajenti
```

1.2 Installation

1.2.1 Debian Packages

Ajenti requires Debian 6 or later. Debian 5 might work with Python 2.6 installed.

Debian Squeeze requires squeeze-backports repository: http://backports.debian.org/Instructions/

Add repository key:

```
wget http://repo.ajenti.org/debian/key -O- | apt-key add -
```

Add repository to /etc/apt/sources.list:

```
echo "deb http://repo.ajenti.org/debian main main debian" >> /etc/apt/sources.list
```

Install the package:

```
apt-get update && apt-get install ajenti
```

Start the service:

```
service ajenti restart
```

1.2.2 Ubuntu Packages

Ajenti requires ubuntu 12.04 Precise Pangolin. Previous releases might work with Python upgraded.

Add repository key:

```
wget http://repo.ajenti.org/debian/key -O- | apt-key add -
```

Add repository to /etc/apt/sources.list:

```
echo "deb http://repo.ajenti.org/ng/debian main main ubuntu" >> /etc/apt/sources.list
```

Install the package:

```
apt-get update && apt-get install ajenti
```

Start the service:

```
service ajenti restart
```

1.2.3 RPM Packages

Ajenti requires EPEL repositories: http://fedoraproject.org/wiki/EPEL

Add repository key:

```
wget http://repo.ajenti.org/ajenti-repo-1.0-1.noarch.rpm
rpm -i ajenti-repo-1.0-1.noarch.rpm
```

Install the package:

```
yum install ajenti
```

Start the service:

```
service ajenti restart
```

Note: Package does not match intended download?

```
yum clean metadata
```

1.2.4 FreeBSD Installation

Prerequisites:

```
cd /usr/ports/devel/py-gevent;  make install clean;
cd /usr/ports/devel/py-lxml;    make install clean;
cd /usr/ports/devel/py-pip;     make install clean;
cd /usr/ports/net/py-ldap2;     make install clean;
cd /usr/ports/security/stunnel; make install clean;
```

Download and install latest Ajenti build from PYPI:

```
pip install ajenti
```

Install rc.d script:

```
wget https://raw.github.com/Eugeny/ajenti/master/packaging/files/ajenti-bsd -O /etc/rc.d/ajenti
```

1.3 Running Ajenti

1.3.1 Starting service

Packages install binary *ajenti-panel* and initscript *ajenti*. You can ensure the service is running:

```
service ajenti restart
```

or:

```
/etc/init.d/ajenti restart
```

Ajenti can be run in a verbose debug mode:

```
ajenti-panel -v
```

The panel will be available on **HTTPS** port **8000** by default. The default username is **root**, and the password is **admin**

1.3.2 Commandline options

- **-c**, **–config <file>** - Use given config file instead of default
- **-v** - Debug/verbose logging
- **-d, –daemon** - Run in background (daemon mode)
- **–set-platform <id>** - Override OS detection

1.3.3 Debugging

Running ajenti with `-v` enables additional logging and Exconsole emergency console (see https://github.com/Eugeny/exconsole).

Exconsole can be triggered by a crash, sending SIGQUIT or pressing `Ctrl-\` on the controlling terminal.

CHAPTER 2

Developers

2.1 Getting Started with Plugin Development

2.1.1 Prerequisites

The following are the absolutely minimal set of software required to build and run Ajenti:

- git
- coffee-script (use NPM)
- lessc (use NPM)

If you don't have CoffeeScript or LESS compiler, you won't be able to make changes to Ajenti CSS/JS files. In this case, download sources from PyPI, which includes compiled CSS/JS resources.

Debian/Ubuntu extras:

- apt-show-versions
- python-dbus (ubuntu)

2.1.2 Setting up

Download the source:

```
git clone git://github.com/Eugeny/ajenti.git
```

(or download them from PyPI: https://pypi.python.org/pypi/ajenti)

Install the dependencies:

```
[sudo] pip install -Ur requirements.txt
```

Launch Ajenti in debug mode:

```
make run
```

Navigate to http://localhost:8000/.

Press Ctrl-at any time to launch an interactive Python shell and Ctrl-D to resume Ajenti.

CoffeeScript and LESS files will be recompiled automatically when you refresh the page; Python code will not. Additional debug information will be available in the console output and browser console.

Ajenti source code includes various example plugins under **Demo** category; their source is available in `ajenti/plugins/test` directory.

2.1.3 Creating new plugin package

New plugins can be placed in both `<source>/ajenti/plugins/` (if you expect inclusion in the source tree) and `/var/lib/ajenti/plugins`.

Each plugin package consists of few Python modules, which contain *`ajenti.api.plugin`* classes (*plugins*). Packages also may contain static files, CoffeeScript and LESS code, and XML user interface layouts:

```
* ajenti
  * plugins
    * test
      * content
        * css
          - 1.less
        * js
          - 2.coffee
        * static
          - 3.png
      * layout
        - 4.xml
      - __init__.py
      - main.py
```

2.1.4 Plugins

To get started, create an empty directory `<source>/ajenti/plugins/test`.

Place a file called `__init__.py` there:

```
from ajenti.api import *
from ajenti.plugins import *


info = PluginInfo(
    title='Test',
    icon=None,
    dependencies=[
        PluginDependency('main'),
    ],
)


def init():
    import main
```

In the same directory, create module `main.py`. The comments explain the concept behind plugins architecture:

```
from ajenti.api import *


@interface
class IShape (object):
    """
    This is an interface, specifying the methods required.
    """
    def number_of_corners(self):
        pass


@plugin
class Square (BasePlugin, IShape):
    """
    A sample implementation, note the inheritance from both BasePlugin (optional but gives extra opt
    """

    def init(self):
        """
        init() methods are automatically called for plugins, maintaining inheritance hierarchy
        """
        print 'Square #%s initialized' % id(self)

    def number_of_corners(self):
        return 4


@plugin
class Circle (BasePlugin, IShape):
    def number_of_corners(self):
        return 0



print 'IShape is implemented by', IShape.get_class()
foo = IShape.get()  # get/create any instance of any IShape implementation
# or, more verbose, IShape.get_class().new()
print 'foo corners:', foo.number_of_corners()

# The instances are by default singleton:
print foo == IShape.get()  # True

# But you can create separate ones:
foo2 = IShape.get_class().new()
print foo == foo2  # False, different instances


for another_foo in IShape.get_all():  # iterate over all possible IShape implementations
    print '\n%s says:' % another_foo, another_foo.number_of_corners()


print IShape.get_instances()  # lists all three active IShape instances
```

Output:

```
IShape is implemented by <class 'ajenti.plugins.test.main.Square'>
Square #24838864 initialized
foo corners: 4
True
Square #24838928 initialized
False

<ajenti.plugins.test.main.Square object at 0x17b02d0> says: 4
<ajenti.plugins.test.main.Circle object at 0x17b0390> says: 0
[<ajenti.plugins.test.main.Square object at 0x17b02d0>, <ajenti.plugins.test.main.Square object at 0x
```

Learn about more interface and plugin methods here: `ajenti.api.plugin`

Continue to *User Interface*

2.2 User Interface

2.2.1 Theory

The whole Ajenti UI is a DOM tree of `ajenti.ui.UIElement` objects. After each update, the UI tree is serialized into JSON and sent to browser, where HTML DOM is assembled from it with the help of CoffeeScript code. Unlike conventional web apps, Ajenti is a stateful machine, which means you adopt a simple workflow similar to developing desktop apps, not websites.

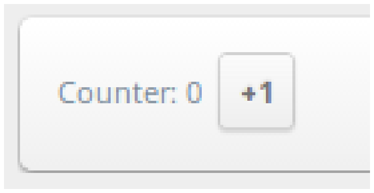
2.2.2 Example

```
from ajenti.api import *
from ajenti.plugins.main.api import SectionPlugin
from ajenti.ui import on


@plugin
class TestPlugin (SectionPlugin):
    def init(self):
        self.title = 'Test'  # those are not class attributes and can be only set in or after init()
        self.icon = 'question'
        self.category = 'Demo'

        """
        UI Inflater searches for the named XML layout and inflates it into
        an UIElement object tree
        """
        self.append(self.ui.inflate('test:main'))

        self.counter = 0
        self.refresh()

    def refresh(self):
        """
        Changing element properties automatically results
        in an UI updated being issued to client
        """
        self.find('counter-label').text = 'Counter: %i' % self.counter

    @on('increment-button', 'click')
    def on_button(self):
        """
        This method is called every time a child element
        with ID 'increment-button' fires a 'click' event
        """
        self.counter += 1
        self.refresh()
```

Add a subdirectory `layout` and place a file named `main.xml` there:

```
<body> <!-- an overall plugin container panel -->
    <pad> <!-- adds whitespace padding -->
        <hc> <!-- horizontal container -->
            <label id="counter-label" />
            <button id="increment-button" text="+1" style="mini" />
        </hc>
    </pad>
</body>
```

Now restart Ajenti. The new plugin **Test** will be visible under **Demo** category. Clicking the **+1** button will increase the counter.



The visible part of plugin is an UIElement, inherited from `ajenti.plugins.main.api.SectionPlugin`.

When you click the button, the 'click' even is fired down the UI tree. The first method to have correctly decorated `@on` method will handle the event. Alternatively, you can set event handler on the element itself by adding this code to `init`:

```
self.find('increment-button').on('click', self.on_button)
```

2.2.3 List of UI Elements

2.2.4 Containers

<box>: Box

Simplest container ever, can be scrollable

```
@p('width', default=None)
@p('height', default=None)
@p('scroll', default=False, type=bool)
```

<pad>: Whitespace

Adds a padding on four sides.

<indent>: Indentation

Adds a padding on two sides.

<right>: Pull-to-right

Pulls its content to right with `float:  right`

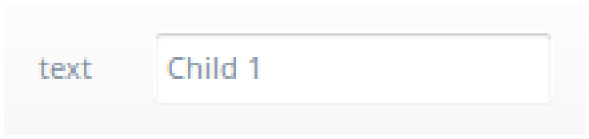
<hc>: Horizontal Container

A horizontal stacking container

<vc>: Vertical Container

A vertical stacking container

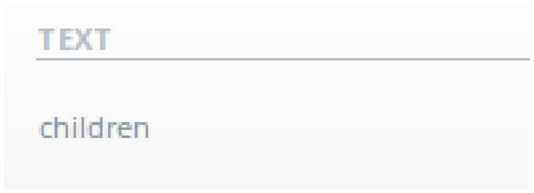
<formline>: Form Line



Container for form controls, has a caption

```
@p('text', default='', bindtypes=[str, unicode])
```

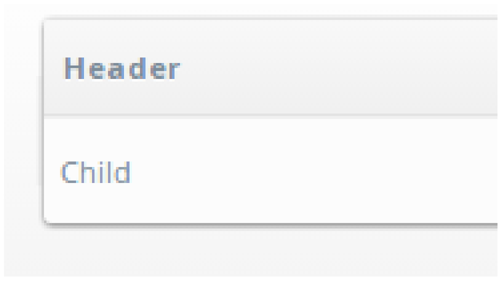
<formgroup>: Form Group



Provides a nice form section separator

```
@p('text', default='', bindtypes=[str, unicode])
```

<dt>, <dtr>, <dth> <dtd>: Data Table



A lined table

```
<dt>
    <dtr>
        <dth text="Header" />
    </dtr>
    <dtr>
        <dtd>
            <label text="Child" />
        </dtd>
    </dtr>
</dt>
```

<collapserow>: Collapsible Table Row

A click-to expand table row

```
<dt>
    <collapserow>
        <label text="Header Child" />
        <label text="Body Child" />
    </collapserow>
</dt>
```

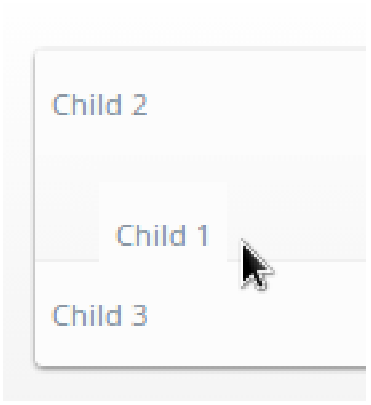
First child is a header and always visible. Second is the collapsible body.

```
@p('expanded', default=False, type=bool, bindtypes=[bool])
```

<lt>, <ltr>, <ltd>: Layout Table

An invisible layout grid (no padding).

<sortabledt>: Sortable Data Table

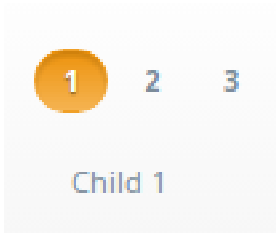


User will be able to reorder rows

```
<sortabledt>
    <dtr>
        <dtd>
            <label text="Child 1" />
        </dtd>
    </dtr>
    <dtr>
        <dtd>
            <label text="Child 2" />
        </dtd>
    </dtr>
    <dtr>
        <dtd>
            <label text="Child 3" />
        </dtd>
    </dtr>
</sortabledt>

@p('sortable', default=True, type=bool)
@p('order', default='', type=str)
```

The **order** property holds the reordered element indexes (`[2,1,3]` as seen on the image)

<tabs>, <tab>: Tabs



User will be able to reorder rows

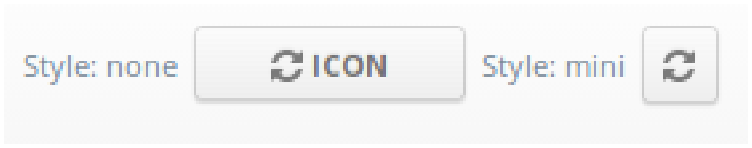
```
<tabs>
    <tab title="1">
        <label text="Child 1" />
    </tab>
    <tab title="2">
        <label text="Child 2" />
    </tab>
    <tab title="3">
        <label text="Child 3" />
    </tab>
</tabs>

<tabs>:
@p('active', default=0)

<title>:
@p('title', default='', bindtypes=[str, unicode])
```

2.2.5 Standard Controls

<label>: Label

@p('text', default='', bindtypes=[str, unicode, int, float])

<button>: Button



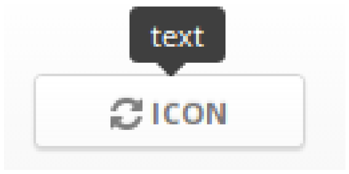
```
@p('text', default='', bindtypes=[str, unicode])
@p('icon', default=None)
@p('warning', default=None)  # display a warning text before click

click()  # fired on click
```

<icon>: Inline Icon

Icon IDs in Ajenti are coming from this page: http://fortawesome.github.io/Font-Awesome/icons/

@p('icon', default=None, bindtypes=[str, unicode])

<tooltip>: Tooltip

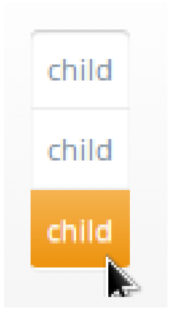


```
@p('text', default='', bindtypes=[str, unicode, int])
```

<progressbar>: Progress Bar

```
@p('width', default=None)
@p('value', default=0, type=float, bindtypes=[float]) # between 0.0 and 1.0
```

<list>, <listitem>: Lists



A list with clickable items:

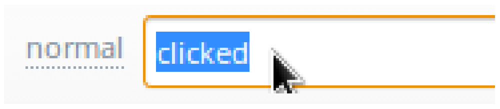
```
<list>
    <listitem>
        <label text="child" />
    </listitem>
    <listitem>
        <label text="child" />
    </listitem>
    <listitem>
        <label text="child" />
    </listitem>
</list>

<listitem>:
click()  # fired on click
```

2.2.6 Inputs

<textbox>: Textbox

```
@p('value', default='', bindtypes=[str, unicode, int])
@p('type', default='text')  # or 'integer'
```

<editable>: Editable Label



A label that becomes textbox when clicked:

```
@p('value', default='', bindtypes=[str, unicode])
@p('icon', default=None)
```

<checkbox>: Checkbox

```
@p('text', default='')
@p('value', default=False, bindtypes=[bool])
```

<dropdown>: Dropdown Select

```
@p('labels', default=[], type=list)
@p('values', default=[], type=list)
@p('value', default='', bindtypes=[str, int, unicode])
```

<combobox>: Combo Box

```
@p('labels', default=[], type=list)
@p('values', default=[], type=list)
@p('separator', default=None, type=str) # if set, combobox becomes autocomplete-multiple-input-box
@p('value', default='', bindtypes=[str, unicode])
```

2.3 Notifications

2.3.1 Example

Code:

```
from ajenti.api import plugin
from ajenti.plugins.main.api import SectionPlugin
from ajenti.ui import on


@plugin
class Test (SectionPlugin):
    def init(self):
        self.title = 'Notifications'
        self.icon = 'smile'
        self.category = 'Demo'

        self.append(self.ui.inflate('test_notifications:main'))
        self.find('style').labels = self.find('style').values = ['info', 'warning', 'error']

    @on('show', 'click')
    def on_show(self):
        self.context.notify(self.find('style').value, self.find('text').value)
```

Layout:

```
<body>
    <pad>
        <vc>
            <formline text="Text">
                <textbox id="text" />
            </formline>
            <formline text="Style">
                <dropdown id="style" />
            </formline>
            <formline>
                <button icon="ok" id="show" text="Show" />
            </formline>
        </vc>
    </pad>
</body>
```

Download this example

2.4 Bindings

Binding mechanism lets you bind your Python objects directly to UI elements and build CRUD interfaces in minutes.

Example: https://github.com/Eugeny/ajenti/blob/dev/ajenti/plugins/test/binder/main.py

2.4.1 Simple bindings

Code:

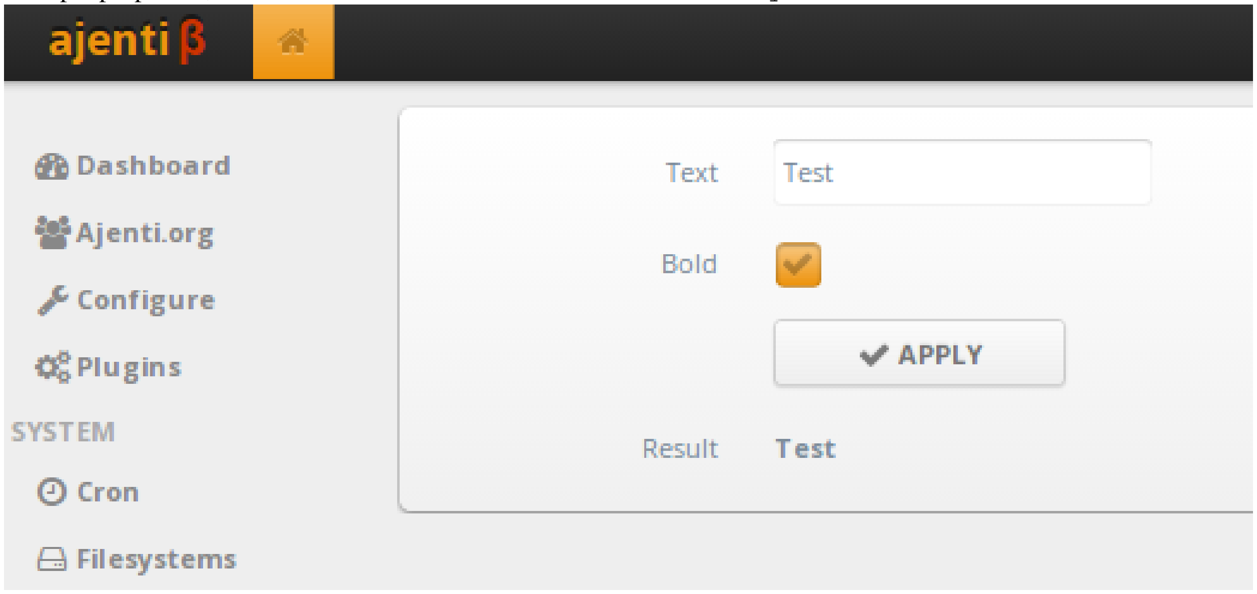
```
from ajenti.api import plugin
from ajenti.plugins.main.api import SectionPlugin
from ajenti.ui import on
from ajenti.ui.binder import Binder


class Settings (object):  # use new-style object at all times!
    def __init__(self):
        self.label_text = ''
        self.label_bold = False
        self.label_style = ''


@plugin
class Test (SectionPlugin):
    def init(self):
        self.title = 'Bindings'
        self.icon = 'smile'
        self.category = 'Demo'

        self.append(self.ui.inflate('test_bindings:main'))

        self.settings = Settings()

        # Bind the settings object to the section UI element (self)
        self.binder = Binder(self.settings, self)
        self.binder.populate()

    @on('apply', 'click')
    def on_apply(self):
        self.binder.update()  # update objects from UI
        self.settings.label_style = 'bold' if self.settings.label_bold else ''
        self.binder.populate()  # update UI with objects
```

Here, the `Settings` object acts as a data model. *`ajenti.ui.binder.Binder`* object connects data with UI. `autodiscover` method scans the UI for bindable elements, `populate` method updates UI with the data from bound objects, and `update` method applies UI changes to objects.

Layout:

```
<body>
    <pad>
        <vc>
            <formline text="Text">
                <textbox bind="label_text" />
            </formline>
            <formline text="Bold">
                <checkbox bind="label_bold" />
            </formline>
            <formline>
                <button icon="ok" id="apply" text="Apply" />
            </formline>
            <formline text="Result">
                <label bind:text="label_text" bind:style="label_style" />
            </formline>
        </vc>
    </pad>
</body>
```

We have added `bind` attributes to the elements which are to be auto-populated with values. If you want to bind multiple properties, use XML attributes like `bind:text` or `bind:style`.



Text: Test

Bold: ✓

Result: Test

2.4.2 Collection Bindings

Ajenti supports following collection bindings:

- Binding iterable to list of elements (*ajenti.ui.binder.ListAutoBinding*)
- Binding dict to key-annotated elements (*ajenti.ui.binder.DictAutoBinding*)
- Binding iterable with a child template (*ajenti.ui.binder.CollectionAutoBinding*)

Code:

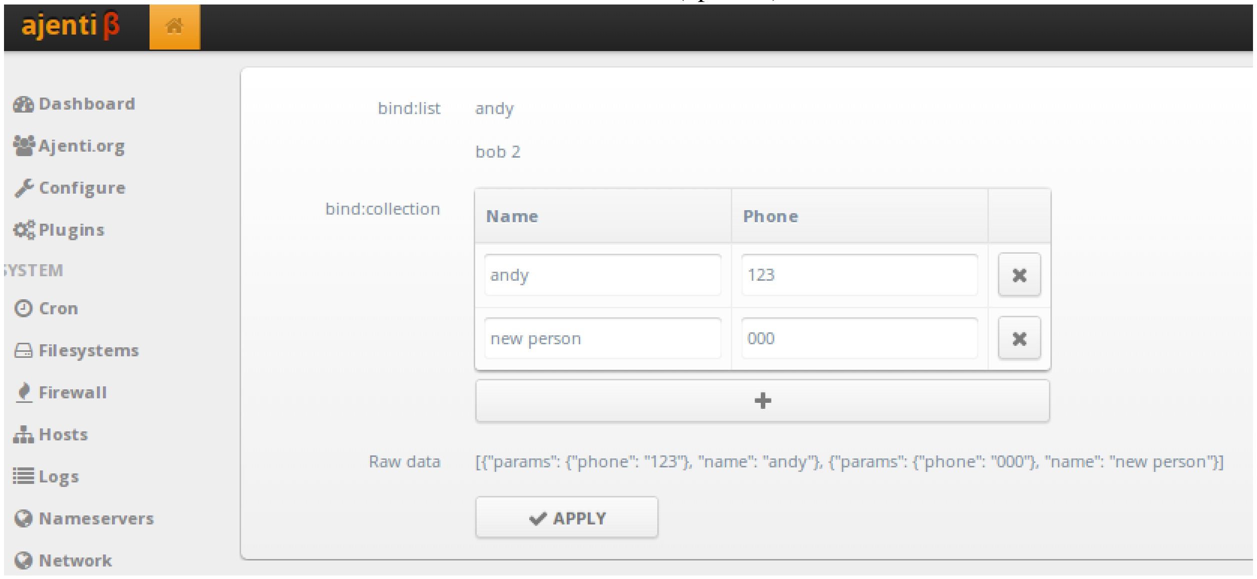
```python
import json

from ajenti.api import plugin
from ajenti.plugins.main.api import SectionPlugin
from ajenti.ui import on
from ajenti.ui.binder import Binder


class Person (object):
    def __init__(self, name, **kwargs):
        self.name = name
        self.params = kwargs

    def __repr__(self):
        return json.dumps({'name': self.name, 'params': self.params})


@plugin
class Test (SectionPlugin):
    def init(self):
        self.title = 'Collection Bindings'
        self.icon = 'smile'
        self.category = 'Demo'
        self.append(self.ui.inflate('test_bindings_collections:main'))

        andy = Person('andy', phone='123')
        bob = Person('bob', phone='321')

        self.obj_list = (andy, bob)
        self.obj_collection = [andy, bob]

        # This callback is used to autogenerate a new item with 'Add' button
        self.find('collection').new_item = lambda c: Person('new person', phone='000')

        self.binder = Binder(self, self)
        self.refresh()

    def refresh(self):
        self.binder.update()
        self.raw_data = repr(self.obj_collection)
        self.binder.populate()

    @on('apply', 'click')
    def on_apply(self):
        self.refresh()
```

Layout:

```
<body>
    <pad>
        <vc>
            <formline text="bind:list">
                <bind:list bind="obj_list">
                    <box>
                        <label bind="name" />
                    </box>
                    <box>
                        <label bind="name" />
                    </box>
                </bind:list>
            </formline>

            <formline text="bind:collection">
                <bind:collection bind="obj_collection" id="collection">
                    <vc>
                        <dt bind="__items">
                            <dtr>
                                <dth text="Name" />
                                <dth text="Phone" />
                                <dth />
                            </dtr>
                        </dt>
                        <button icon="plus" style="mini" bind="__add" />
                    </vc>

                    <bind:template>
                        <dtr>
                            <dtd> <textbox bind="name" /> </dtd>

                            <dtd>
                                <bind:dict bind="params">
                                    <textbox bind="phone" />
                                </bind:dict>
                            </dtd>

                            <dtd> <button icon="remove" style="mini" bind="__delete" /> </dtd>
                        </dtr>
                    </bind:template>

                </bind:collection>
            </formline>

            <formline text="Raw data">
                <label bind="raw_data" />
            </formline>

            <formline>
                <button icon="ok" id="apply" text="Apply" />
            </formline>
        </vc>
    </pad>
</body>
```

Note the special `bind` attribute values used in `bind:collection`:

- `__items` denotes the container for items
- `__add` denotes a button which will generate a new item (optional)
- `__remove` denotes a button which will remove an item (optional)



[{"params": {"phone": "123"}, "name": "andy"}, {"params": {"phone": "000"}, "name": "new person"}]

2.5 Custom UI Controls

You can create any type of a reusable UI control. Remember to take a look at default controls in `ajenti/plugins/main` for guidance.

2.5.1 Example

In this example, we'll create a HTML5 slider control.

Code:

```python
from ajenti.api import plugin
from ajenti.plugins.main.api import SectionPlugin
from ajenti.ui import on, p, UIElement


@plugin
class Test (SectionPlugin):
    def init(self):
        self.title = 'Controls'
        self.icon = 'smile'
        self.category = 'Demo'
        self.append(self.ui.inflate('test_controls:main'))

    @on('check', 'click')
    def on_show(self):
        self.context.notify('info', 'Value is %i' % self.find('slider').value)


@p('value', type=int, default=0)
@plugin
class Slider (UIElement):
    typeid = 'slider'
```

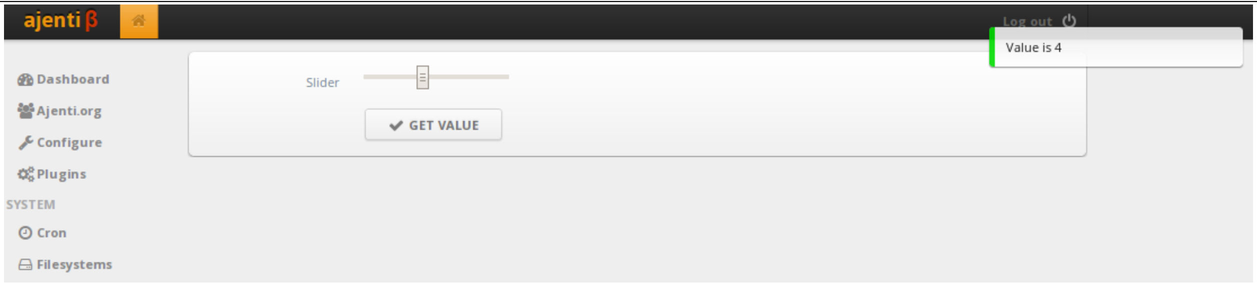
Layout:

```
<body>
    <pad>
        <vc>
            <formline text="Slider">
                <slider id="slider" value="0" />
            </formline>
            <formline>
                <button icon="ok" id="check" text="Get value" />
            </formline>
        </vc>
    </pad>
</body>
```

Control class is decorated with *ajenti.ui.p()* for each of its properties. The main client-side logic is implemented through CoffeeScript code (though you can try to get away with pure-JS).

CoffeeScript:

```
class window.Controls.slider extends window.Control
    createDom: () ->
        # createDom() must return HTML
        """
            <div>
                <input type="range" min="0" max="10" />
            </div>
        """

    setupDom: (dom) ->
        # setupDom may attach event handler and perform other DOM manipulations
        # use this.properties hash to populate control with its current state
        super(dom)
        @input = $(@dom).find('input')
        @input.val(@properties.value)

    detectUpdates: () ->
        # detectUpdates() should return a hash containing only changed properties
        # be sure to not report unchanged properties since this will lead to infinite update loops
        r = {}
        value = parseInt(@input.val())
        if value != @properties.value
            r.value = value
        return r
```



Download this example

2.6 Handling HTTP Requests

2.6.1 Example

This example illustrates various HTTP responses. Try following URLs:

- http://localhost:8000/ajenti:demo/notify?text=hello
- http://localhost:8000/ajenti:demo/respond/redirect
- http://localhost:8000/ajenti:demo/respond/server_error
- http://localhost:8000/ajenti:demo/respond/ok
- http://localhost:8000/ajenti:demo/respond/file

Code:

```
from ajenti.api import plugin, BasePlugin
from ajenti.api.http import HttpPlugin, url


@plugin
class HttpDemo (BasePlugin, HttpPlugin):
    @url('/ajenti:demo/notify')
    def get_page(self, context):
        if context.session.identity is None:
            context.respond_redirect('/')
        self.context.notify('info', context.query.getvalue('text', ''))
        context.respond_ok()
        return ''
    @url('/ajenti:demo/respond/(?P<what>.+)')
    def get_response(self, context, what=None):
        if what == 'ok':
            context.respond_ok()
            return 'Hello!'
        if what == 'redirect':
            return context.respond_redirect('/')
        if what == 'server_error':
            return context.respond_server_error()
        if what == 'forbidden':
            return context.respond_forbidden()
        if what == 'not_found':
            return context.respond_not_found()
        if what == 'file':
            return context.file('/etc/issue')
```

Download this example

CHAPTER 3

API Reference

3.1 ajenti

`ajenti.`**`config`** = **None**
: Loaded config, is a *reconfigure.items.ajenti.AjentiData*

`ajenti.`**`platform`** = **None**
: Current platform

`ajenti.`**`platform_string`** = **None**
: Human-friendly platform name

`ajenti.`**`platform_unmapped`** = **None**
: Current platform without “Ubuntu is Debian”-like mapping

`ajenti.`**`installation_uid`** = **None**
: Unique installation ID

`ajenti.`**`version`** = **None**
: Ajenti version

`ajenti.`**`server`** = **None**
: Web server

`ajenti.`**`debug`** = **False**
: Debug mode

`ajenti.`**`init`**`()`

`ajenti.`**`exit`**`()`

`ajenti.`**`restart`**`()`

3.2 ajenti.api

class `ajenti.api.`**`PluginInfo`**`(`*****kwargs***`)`
: Describes a loaded plugin package

class `ajenti.api.`**`BasePlugin`**
: A base plugin class that provides *`AppContext`* and `classconfig` functionality.

  `classconfig_editor` = **None**
  : Override this in your class with an ajenti.plugins.configurator.api.ClassConfigEditor derivative

classconfig_name = None
: Override this in your class if you want this plugin to be configurable through Configure > Plugins

classconfig_root = False
: When True, classconfig will be stored in root's config section disregarding current user

context = None
: Automatically receives a reference to the current *AppContext*

create_classconfig()

default_classconfig = None
: Override this in your class with a default config object (must be JSON-serializable)

init()
: Do your initialization here. Correct bottom-to-up inheritance call order guaranteed.

load_classconfig()
: Loads the content of `classconfig` attribute from the user's configuration section.

open_content(*path*, *mode='r'*)
: Provides access to plugin-specific files from `/content` directory of the package

Parameters

- **path** (*str*) – path relative to package's `/content`
- **mode** (*str*) – Python file access mode

Returns An open file object

Return type file

save_classconfig()
: Saves the content of `classconfig` attribute into the user's configuration section.

class ajenti.api.**AppContext**(*parent*, *httpcontext*)
: A session-specific context provided to everyone who inherits *BasePlugin*.

session
: current HTTP session: `ajenti.middleware.Session`

user
: current logged in user: `reconfigure.items.ajenti.UserData`

Methods injected by MainPlugin:

notify(*text*)

Parameters **text** – Notification text to show

launch(*id*, **args*, ***kwargs*)

Parameters **id** – Intent ID to be launched

ajenti.api.**plugin**(*cls*)
: A decorator to create plugin classes:

```
@plugin
class SomePlugin (ISomething):
    pass
```

If the class has a `verify` method returning `bool`, it's invoked. If the method returned `False`, plugin is rejected and removed from implementation lists.

If the class has a `platforms` attribute, which is a list of supported platform names, it's compared against the current runtime OS platform. If the current platform is not in the list, plugin is also rejected.

Following class methods are injected.

`.get`(*context=<current context>*)

Returns any existing instance or creates a new one

`.new`(*args, *context=<current context>*, **kwargs)

Returns a new instance. Use this method instead of constructor, since it invokes the proper initialization chain and registers the instance

Return type class, None

`ajenti.api.`**`rootcontext`**(*cls*)

Enforces use of root PluginContext by default for .get() and .new() classmethods.

`ajenti.api.`**`notrack`**(*cls*)

Disables instance tracking of plugin (and derivative) instances within PluginContext via get/get_all and similar methods.

Return type class

`ajenti.api.`**`notrack_this`**(*cls*)

Disables instance tracking of plugin instances within PluginContext via get/get_all and similar methods.

Return type class

`ajenti.api.`**`track`**(*cls*)

Enables previously disabled instance tracking of plugin.

Return type class

`ajenti.api.`**`persistent`**(*cls*)

Makes this plugin non-GCable

Return type class

`ajenti.api.`**`extract_context`**()

An utility function that extracts and returns the nearest *`AppContext`* from the current call stack.

Return type *`ajenti.plugins.PluginContext`*, None

exception `ajenti.api.`**`NoImplementationsError`**

`ajenti.api.`**`interface`**(*cls*)

A decorator to create plugin interfaces:

```
@interface
class ISomething (object):
    def contract(self):
        pass
```

Following class methods are injected:

`.get`(*context=<current context>*)

Returns any existing instance or creates a new one

`.get_all`(*context=<current context>*)

Returns list of instances for each implementation

`.get_class`()

Returns any implementation class

`.get_classes()`

Returns list of implementation classes

`.get_instances`(*context=<current context>*)

Returns list of all existing instances

Return type class

3.3 ajenti.api.http

3.4 ajenti.api.sensors

class `ajenti.api.sensors.`**`Sensor`**

Base class for a Sensor. Sensors measure system status parameters and can be queried from other plugins.

static **`find`**(*id*)

Returns a Sensor by name

Parameters **`id`** (*str*) – sensor ID

Return type *`Sensor`*, None

`get_variants`()

Override this and return a list of available variants.

Return type list

`id` **= None**

`init`()

`measure`(*variant=None*)

Override this and perform the measurement.

Parameters **`variant`** (*str, None*) – variant to measure

Return type int, float, tuple, list, dict, str

`timeout` **= 0**

`value`(*variant=None*)

Returns sensor's measurement for a specific *variant*. Sensors can have multiple variants; for example, disk usage sensor accepts device name as a variant.

Parameters **`variant`** (*str, None*) – variant to measure

Return type int, float, tuple, list, dict, str

3.5 ajenti.http

3.6 ajenti.ipc

3.7 ajenti.middleware

3.8 ajenti.plugins

exception `ajenti.plugins.`**`PluginLoadError`**

exception `ajenti.plugins.`**`PluginFormatError`**

> **`describe`**`()`

exception `ajenti.plugins.`**`PluginCrashed`**`(`*e*`)`

> **`describe`**`()`

class `ajenti.plugins.`**`Dependency`**

> **exception** **`Unsatisfied`**
>
> > **`describe`**`()`
> >
> > **`reason`**`()`
>
> `Dependency.`**`build_exception`**`()`
>
> `Dependency.`**`check`**`()`
>
> `Dependency.`**`satisfied`**`()`
>
> `Dependency.`**`value`**

class `ajenti.plugins.`**`ModuleDependency`**`(`*module_name*`)`

> **exception** **`Unsatisfied`**
>
> > **`reason`**`()`
>
> `ModuleDependency.`**`description`** **= ‘Python module’**
>
> `ModuleDependency.`**`is_satisfied`**`()`

class `ajenti.plugins.`**`PluginDependency`**`(`*plugin_name*`)`

> **exception** **`Unsatisfied`**
>
> > **`reason`**`()`
>
> `PluginDependency.`**`description`** **= ‘Plugin’**
>
> `PluginDependency.`**`is_satisfied`**`()`

class `ajenti.plugins.`**`BinaryDependency`**(*binary_name*)

> **exception** **`Unsatisfied`**
>
> > **`reason`**()
>
> `BinaryDependency.`**`description`** **= 'Application binary'**
>
> `BinaryDependency.`**`is_satisfied`**()

class `ajenti.plugins.`**`FileDependency`**(*file_name*)

> **exception** **`Unsatisfied`**
>
> > **`reason`**()
>
> `FileDependency.`**`description`** **= 'File'**
>
> `FileDependency.`**`is_satisfied`**()

class `ajenti.plugins.`**`PluginContext`**

> Container of interfaces and plugins
>
> **`get_instance`**(*cls*)
>
> **`get_instances`**(*cls*)
>
> **`instantiate`**(*cls*, **args*, ***kwargs*)
>
> **`vacuum_instances`**()

class `ajenti.plugins.`**`PluginManager`**

> Handles plugin loading and unloading
>
> **`blacklist`** **= []**
>
> **`context`** **= <ajenti.plugins.PluginContext object>**
>
> **`extra_location`** **= '/var/lib/ajenti/plugins'**
>
> **`get_all`**()
>
> **`get_implementations`**(*iface*)
>
> **`get_order`**()
>
> **`get_plugins_root`**()
>
> **`load`**(*name*)
>
> > Loads given plugin
>
> **`load_all`**()
>
> **`load_recursive`**(*name*)
>
> **`register_implementation`**(*impl*)
>
> **`register_interface`**(*iface*)
>
> **`resolve_path`**(*name*)

3.9 ajenti.profiler

`ajenti.profiler.`**`get_profiles`**`()`
  Returns all accumulated profiling values

`ajenti.profiler.`**`profile_end`**`(`*name=None*`)`
  Ends a profiling interval with specific `name`

`ajenti.profiler.`**`profile_start`**`(`*name*`)`
  Starts a profiling interval with specific `name` Profiling data is sent to the client with next data batch.

`ajenti.profiler.`**`profiled`**`(`*namefx=None*`)`

3.10 ajenti.ui

class `ajenti.ui.`**`UI`**
  The root UI object, one per session

  `clear_updates``()`
    Marks all pending updates as processed

  `create``(`*typeid*, **args*, ***kwargs*`)`
    Creates an element by its type ID.

      Parameters **`typeid`** (*str*) – type ID

  `dispatch_event``(`*uid*, *event*, *params=None*`)`
    Dispatches an event to an element with given UID

      Parameters

      - **`uid`** (*int*) – element UID
      - **`event`** (*str*) – event name
      - **`params`** (*dict, None*) – event arguments

  `find``(`*id*`)`

      Parameters **`id`** (*str*) – element ID

      Returns nearest element with given ID

      Return type UIElement, None

  `find_uid``(`*uid*`)`

      Parameters **`uid`** (*int*) – element UID

      Returns nearest element with given unique ID

      Return type UIElement, None

  `has_updates``()`
    Checks for pending UI updates

      Return type bool

  `inflate``(`*layout*`)`

      Parameters **`layout`** (*str*) – layout spec: “<plugin id>:<layout file name without extension>”

      Returns an inflated element tree of the given layout XML name

Return type *UIElement*

init()

render()
Renders the UI into JSON

Return type dict

class ajenti.ui.**UIElement**(*ui*, *typeid=None*, *children=[]*, ***kwargs*)
Base UI element class

append(*child*)
Appends a `child`

Parameters **child** (*UIElement*) – child

bind
Bound property name

bindtransform
Value transformation function for one-direction bindings

broadcast(*method*, **args*, ***kwargs*)
Calls `method` on every member of the subtree

Parameters **method** (*str*) – method

clear_updates()
Marks all pending updates as processed

client
Whether this element's events are only processed on client side

clone(*set_ui=None*, *set_context=None*)

Returns a deep copy of the element and its children. Property values are shallow copies.

Return type `UIElement`

contains(*element*)
Checks if the `element` is in the subtree of `self`

Parameters **element** (*UIElement*) – element

delete()
Detaches this element from its parent

dispatch_event(*uid*, *event*, *params=None*)
Dispatches an event to an element with given UID

Parameters

- **uid** (*int*) – element UID
- **event** (*str*) – event name
- **params** (*dict, None*) – event arguments

empty()
Detaches all child elements

event(*event*, *params=None*)
Invokes handler for `event` on this element with given `**params`

Parameters

- **`event`** (*str*) – event name
- **`params`** (*dict, None*) – event arguments

`find`(*id*)

Parameters **`id`** (*str*) – element ID

Returns the nearest child with given ID or `None`

Return type *`UIElement`*, None

`find_type`(*typeid*)

Returns the nearest child with given type ID or `None`

Return type *`UIElement`*, None

`find_uid`(*uid*)

Parameters **`uid`** (*int*) – element UID

Returns the nearest child with given UID or `None`

Return type *`UIElement`*, None

`has_updates`()

Checks for pending UI updates

`id`

Element ID

`init`()

`invalidate`()

`nearest`(*predicate*, *exclude=None*, *descend=True*)

Returns the nearest child which matches an arbitrary predicate lambda

Parameters

- **`predicate`** (*function*) – `lambda element: bool`
- **`exclude`** (*function, None*) – `lambda element: bool` - excludes matching branches from search
- **`descend`** (*bool*) – whether to descend inside matching elements

`on`(*event*, *handler*, **args*)

Binds event with ID `event` to `handler`. `*args` will be passed to the `handler`. :param event: event :type event: str :param handler: handler :type handler: function

`path_to`(*element*)

Returns a list of elements forming a path from `self` to `element`

Return type list

`post_clone`()

`property_definitions`

`remove`(*child*)

Detaches the `child`

Parameters **`child`** (*`UIElement`*) – child

`render`()

Renders this element and its subtree to JSON

Return type dict

`reverse_event` (*event*, *params=None*)
Raises the event on this element by feeding it to the UI root (so that `@on` methods in ancestors will work).

Parameters

- **`event`** (*str*) – event name
- **`params`** (*dict*) – event arguments

`style`
Additional CSS class

`typeid` = **None**

`visible`
Visibility of the element

`ajenti.ui.`**`p`**(*prop*, *default=None*, *bindtypes=[]*, *type=<type 'unicode'>*, *public=True*, *doc=None*)
Creates an UI property inside an *`UIElement`*:

```
@p('title')
@p('category', default='Other', doc='Section category name')
@p('active', default=False)
class SectionPlugin (BasePlugin, UIElement):
    typeid = 'main:section'
```

Parameters

- **`default`** (*object*) – Default value
- **`bindtypes`** (*list*) – List of Python types that can be bound to this property
- **`type`** (*object*) – expected Python type for this value
- **`public`** (*bool*) – whether this property is rendered and sent to client
- **`doc`** (*str, None*) – docstring

Return type function

`ajenti.ui.`**`on`**(*id*, *event*)
Sets the decorated method to handle indicated event:

```
@plugin
class Hosts (SectionPlugin):
    def init(self):
        self.append(self.ui.inflate('hosts:main'))
        ...

    @on('save', 'click')
    def save(self):
        self.config.save()
```

Parameters

- **`id`** (*str*) – element ID
- **`event`** (*str*) – event name

Return type function

3.11 ajenti.ui.binder

class `ajenti.ui.binder.`**`Binding`**(*object*, *attribute*, *ui*)

A base class for bindings. Binding is a link between a Python object attribute and Ajenti UI element's property.

Parameters

- **`object`** – a Python object
- **`attribute`** – attribute name
- **`ui`** – Ajenti *`ajenti.ui.UIElement`*

classmethod **`applicable`**(*object*, *attribute*)

classmethod **`extract`**(*object*, *attribute*, *ignore_errors=True*)

`get`()

Returns value of the bound attribute

`populate`()

Should update the UI with attribute's value

`set`(*value*)

Sets value of the bound attribute

`unpopulate`()

Should revert UI to normal state

`update`()

Should update the attribute with data from the UI

class `ajenti.ui.binder.`**`PropertyBinding`**(*obj*, *attribute*, *ui*, *property=None*)

A simple binding between UI element's property and Python object's attribute

Parameters **`property`** – UI property name. If `None`, property is deduced from `bindtypes`

`populate`()

`update`()

class `ajenti.ui.binder.`**`ListAutoBinding`**(*object*, *attribute*, *ui*)

Binds values of a collection to UI element's children consecutively, using *`Binder`*

`populate`()

`unpopulate`()

`update`()

class `ajenti.ui.binder.`**`DictAutoBinding`**(*object*, *attribute*, *ui*)

Binds values from a dict to UI element's children mapping 'bind' attribute to dict key, using *`Binder`*

`populate`()

`unpopulate`()

`update`()

class `ajenti.ui.binder.`**`CollectionAutoBinding`**(*object*, *attribute*, *ui*)

Binds values of a collection to UI element's children using a template. The expected UI layout:

```
<xml xmlns:bind="bind">
    <bind:collection id="<binding to this>">
        <container-element bind="__items">
            <1-- instantiated templates will appear here -->
        </container-element>

        <bind:template>
            <!-- a template for one collection item
                 it will be bound to item using ajenti.ui.binder.Binder -->
            <label bind="some_property" />

            <button id="__delete" /> <!-- a delete button may appear in the template -->
        </bind:template>

        <button id="__add" /> <!-- an add button may appear inside collection tag -->
    </bind:collection>
</xml>
```

get_template(*item*, *ui*)

on_add()

on_delete(*item*)

populate()

set_page(*page=0*)

unpopulate()

update()

class ajenti.ui.binder.**Binder**(*object=None*, *ui=None*)
An automatic object-to-ui-hierarchy binder. Uses `bind` UI property to find what and where to bind. If `object` is not None, the Binder is also initialized (see `setup(object)`) with this data object.

Parameters

- **object** – Python object
- **ui** – UI hierarchy root

add(*binding*)

autodiscover(*object=None*, *ui=None*)

populate()
Populates the bindings.

reset(*object=None*, *ui=None*)
Cancels the binding and replaces Python object / UI root.

setup(*object=None*)
Initializes the Binder with a data object. :type object: object

unpopulate()
Unpopulates the bindings.

update()
Updates the bindings.

class ajenti.ui.binder.**BasicCollectionElement**(*ui*, *typeid=None*, *children=[]*, ***kwargs*)

binding
: Collection binding class to use

filter
: Called to filter collections values, `lambda value: bool`

post_bind
: Called after binding is complete, `lambda object, collection, ui: None`

post_item_bind
: Called after an item is bound, `lambda object, collection, item, item-ui: None`

post_item_update
: Called after an item is updated, `lambda object, collection, item, item-ui: None`

values
: Called to extract values from the collection, `lambda collection: []`

class `ajenti.ui.binder.`**`ListElement`**(*ui*, *typeid=None*, *children=[]*, ***kwargs*)

typeid = 'bind:list'

class `ajenti.ui.binder.`**`CollectionElement`**(*ui*, *typeid=None*, *children=[]*, ***kwargs*)

add_item
: Called to append value to the collection, `lambda item, collection: None`

delete_item
: Called to remove value from the collection, `lambda item, collection: None`

new_item
: Called to create an empty new item, `lambda collection: object()`

pagesize

sorting
: If defined, used as key function to sort items

typeid = 'bind:collection'

3.12 ajenti.users

`ajenti.users.`**`restrict`**(*permission*)
: Marks a decorated function as requiring `permission`. If the invoking user doesn't have one, *`SecurityError`* is raised.

class `ajenti.users.`**`PermissionProvider`**
: Override to create your own set of permissions

get_name()
: Should return a human-friendly name for this set of permissions (displayed in Configurator) :rtype: str

get_permissions()
: Should return a list of permission names

Return type list

exception `ajenti.users.`**`SecurityError`**(*permission*)
: Indicates that user didn't have a required permission.

permission
permission ID

class ajenti.users.**UserManager**

check_password(*username*, *password*, *env=None*)
Verifies the given username/password combo

Return type bool

classconfig_root = True

default_classconfig = {'sync-provider': ''}

get_sync_provider(*fallback=False*)

Return type ajenti.usersync.UserSyncProvider

has_permission(*context*, *permission*)
Checks whether the current user has a permission

Return type bool

hash_password(*password*)

Return type str

hash_passwords()

require_permission(*context*, *permission*)
Checks current user for given permission and raises *SecurityError* if he doesn't have one :type permission: str :raises: SecurityError

set_password(*username*, *password*)

set_sync_provider(*provider_id*)

3.13 ajenti.util

ajenti.util.**public**(*f*)
" Use a decorator to avoid retyping function/class names.

Based on an idea by Duncan Booth: http://groups.google.com/group/comp.lang.python/msg/11cbb03e09611b8a

Improved via a suggestion by Dave Angel: http://groups.google.com/group/comp.lang.python/msg/3d400fb22d8a42e1

ajenti.util.**str_fsize**(*sz*)
Formats file size as string (i.e., 1.2 Mb)

ajenti.util.**str_timedelta**(*s*)
Formats a time delta (i.e., "5 days, 5:06:07")

ajenti.util.**cache_value**(*duration=None*)
Makes a function lazy.

Parameters **duration** (*int*) – cache duration in seconds (default: infinite)

ajenti.util.**platform_select**(***values*)
Selects a value from **kwargs** depending on runtime platform

```
service = platform_select(
    debian='samba',
    ubuntu='smbd',
    centos='smbd',
    default='samba',
)
```

`ajenti.util.`**`make_report`**`(`*e*`)`

Formats a bug report.

CHAPTER 4

Plugin API Reference

4.1 ajenti.plugins.main.api

4.2 ajenti.plugins.dashboard.api

class `ajenti.plugins.dashboard.api.`**`ConfigurableWidget`**(*ui*, *typeid=None*, *children=[]*, ***kwargs*)

Base class for widgets with a configuration dialog

`begin_configuration`()

`create_config`()
Should return a default config dict

`init`()

`on_config`(*button*)

`on_config_save`()
Called when user is done configuring the widget.

`on_config_start`()
Called when user begins to configure the widget. Should populate the config dialog.

`on_prepare`()
Widget should create its UI in this method. Called before **self.config** is created

`on_start`()
Widget should populate its UI in this method. **self.config** is now available.

class `ajenti.plugins.dashboard.api.`**`DashboardWidget`**(*ui*, *typeid=None*, *children=[]*, ***kwargs*)

Base class for widgets (inherits `ajenti.ui.UIElement`).

`config`
current configuration dict of this widget instance

`container`

`hidden` **= False**
If True, user will not be able to add this widget through dashboard

`icon` **= None**
Widget icon name

`index`

name = '—'

Widget type name

save_config()

typeid = 'dashboard:widget'

4.3 ajenti.plugins.configurator.api

class ajenti.plugins.configurator.api.**ClassConfigEditor**(*ui*, *typeid=None*, *children=[]*, ***kwargs*)

title

typeid = 'configurator:classconfig-editor'

4.4 ajenti.plugins.db_common.api

4.5 ajenti.plugins.webserver_common.api

4.6 ajenti.plugins.packages.api

class ajenti.plugins.packages.api.**PackageInfo**

class ajenti.plugins.packages.api.**PackageManager**

do(*actions*, *callback=<function <lambda>>*)

get_lists()

init()

refresh()

search(*query*)

4.7 ajenti.plugins.services.api

class ajenti.plugins.services.api.**Service**

command(*cmd*)

icon

restart()

source = 'unknown'

Marks which ServiceManager owns this object

start()

stop()

class `ajenti.plugins.services.api.`**`ServiceManager`**

`get_one`(*name*)
Returns a *`Service`* by name.

class `ajenti.plugins.services.api.`**`ServiceMultiplexor`**
Merges together output of all available ServiceManagers.

`get_all`(**args*, ***kwargs*)
Returns all *`Service`* s.

`get_one`(*name*)
Returns a *`Service`* by name.

`init`()

4.8 ajenti.plugins.tasks.api

class `ajenti.plugins.tasks.api.`**`JobDefinition`**(*j={}*)

`save`()

class `ajenti.plugins.tasks.api.`**`Task`**(***kwargs*)
Base class for custom tasks

Parameters

- **`name`** – display name
- **`ui`** – full layout name for parameter editor, will be bound to parameter dictionary (so begin it with <bind:dict bind="params">)
- **`hidden`** – if True, task won't be available for manual creation

`abort`()

`get_progress`()

`hidden` **= False**

`init`()

`name` **= '—'**

`run`(***kwargs*)
Override with your task actions here. Raise *`TaskError`* in case of emergency. Check *aborted* often and return if it's True

`set_progress`(*current*, *max*)

`start`()

`ui` **= None**

class `ajenti.plugins.tasks.api.`**`TaskDefinition`**(*j={}*, *task_class=None*)

`get_class`()

`save`()

exception `ajenti.plugins.tasks.api.`**`TaskError`**

class `ajenti.plugins.tasks.api.`**`TaskResult`**

> **`ABORTED`** = **1**
>
> **`CRASH`** = **3**
>
> **`ERROR`** = **2**
>
> **`SUCCESS`** = **0**

CHAPTER 5

Indices and tables

- genindex
- modindex
- search