Ajenti

Release 2.1.18

May 26, 2017

Users

Ajenti platform includes following products:

- **Ajenti Core**, a Python library, the platform itself including the HTTP server, socket engine and plugin container.
- **Ajenti Panel**, a startup script and a set of stock plugins such as file manager, network configurator and service manager.

CHAPTER 1

Feature Overview

HTTP Server

- HTTP 1.1 Support.
- Websockets with fallback to XHR polling.
- Fast event-loop based processing.
- Flexible routing.
- Session sandboxing.
- SSL with client certificate authentication.

Performance

- >1000 requests per second.
- 30 MB RAM footprint + 5 MB per session.

API

- Highly modular Python API. Everything is a module and can be removed or replaced.
- Builtin webserver API supports routing, file downloads, GZIP, websockets and more.
- Transparent SSL client authorization.
- Plugin architecture
- Dependency injection
- Server-side push and socket APIs.

Security

- Pluggable authentication and authorization.
- Stock authenticators: UNIX account, password, SSL client certificate and Mozilla Persona E-mail authentication.
- Unprivileged sessions isolated in separate processes.

Frontend

- Clean, modern and responsive UI. Single-page, no reloads.
- Live data updates and streaming with Socket.IO support.
- Full mobile and tablet support.
- LESS and CSS, CoffeeScript and JavaScript auto-build support.
- Numerous stock directives.
- AngularJS templating.

Platforms

- Debian 6 or later
- Ubuntu Precise or later
- CentOS 6 or later
- RHEL 6 or later
- Can be run on other Linux or BSD systems with minimal modifications.
- Supports Python 2.7+ and PyPy.

Installing

Caution: Supported operating systems:

- Debian 6 or later
- Ubuntu Precise or later
- CentOS 6 or later
- RHEL 6 or later

Other Linux-based systems *might* work, but you'll have to use manual installation method.

Automatic Installation

```
curl https://raw.githubusercontent.com/ajenti/ajenti/master/scripts/install.sh | sudo␣
↪bash -s -
```

Manual Installation

Native dependencies: Debian/Ubuntu

```
sudo apt-get install build-essential python-pip python-dev python-lxml libffi-dev␣
↪libssl-dev libjpeg-dev libpng-dev uuid-dev python-dbus
```

Native dependencies: RHEL/CentOS

```
sudo yum install gcc python-devel python-pip libxslt-devel libxml2-devel libffi-devel␣
↪openssl-devel libjpeg-turbo-devel libpng-devel dbus-python
```

Install Ajenti

Upgrade PIP:

```
sudo pip install 'setuptools>=0.6rc11' 'pip>=6' wheel
```

Minimal install:

```
sudo pip install ajenti-panel ajenti.plugin.dashboard ajenti.plugin.settings ajenti.
↪plugin.plugins
```

With more plugins:

```
sudo pip install ajenti-panel ajenti.plugin.dashboard ajenti.plugin.settings ajenti.
↪plugin.plugins ajenti.plugin.filemanager ajenti.plugin.notepad ajenti.plugin.
↪packages ajenti.plugin.services ajenti.plugin.terminal
```

Running Ajenti

Starting service

Ajenti provides binary *ajenti-panel* and initscript/job/unit *ajenti*. You can ensure the service is running:

```
service ajenti restart
```

or:

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

or:

```
systemctl restart ajenti
```

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

Ajenti can also be run in a verbose debug mode:

```
ajenti-panel -v
```

Commandline options

- `-c`, `--config <file>` - Use given config file instead of default
- `-v` - Debug/verbose logging
- `--dev` - Enables automatic resources build on each request
- `-d, --daemon` - Run in background (daemon mode)
- `--autologin` - Will automatically log in the user under which the panel runs. **This is a security issue if your system is public**.

Debugging

Ajenti logs into `/var/log/ajenti/ajenti.log`.

Running ajenti with `-v` enables additional logging.

What's Ajenti and how it works

Ajenti Project itself consists of **Ajenti Core** itself and a set of stock plugins forming the **Ajenti Panel**.

Ajenti Core

Ajenti Core is a web interface development framework which includes a web server, IoC container, a simplistic web framework and set of core components aiding in client-server communications.

Ajenti Panel

Ajenti Panel consists of plugins developed for the Ajenti Core and a startup script, together providing a server administration panel experience. The Panel's plugins include: file manager, terminal, notepad, etc.

Modus operandi

During bootstrap, Ajenti Core will locate and load Python modules containing Ajenti plugins (identified by a `plugin.yml` file). It will then register the implementation classes found in them in the root IoC container. Some interfaces to be implemented include *aj.api.http.HttpPlugin*, *aj.plugins.core.api.sidebar.SidebarItemProvider*.

Ajenti Core runs a HTTP server on a specified port, managing a pool of isolated session workers and forwarding requests to these workers, delivering them to the relevant *aj.api.http.HttpPlugin* instances. It also supports Socket.IO connections, forwarding them to the relevant *aj.api.http.SocketEndpoint* instances.

Ajenti contains a mechanism for session authentication through PAM login and `sudo` elevation. Standard `core` plugin provides HTTP API for that.

Authenticated sessions are moved to isolated worker processes running under the corresponding account.

Ajenti frontend is an AngularJS application composed from Angular modules provided by each plugin. Every plugin can contribute its own JS/CSS code to the combined resource package delivered to the client.

The `core` plugin provides a `ng:view` container for `ngRoute` navigation. So, the plugins that have UI are expected to provide additional `ngRoute` routes, templates and controllers.

Getting Started

Required knowledge

- Python 2
- JavaScript (ES5, ES6 or CoffeeScript)
- basic AngularJS knowledge (modules & controllers)
- basic HTML skills

Setting up development environent

1. Install Ajenti

We recommend to use the automatic installer - see the *installation guide*

2. Install build tools

Build tools require NodeJS - you can use the NodeSource repositories for quick setup:

```
# Using Ubuntu
curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
sudo apt-get install -y nodejs

# Using Debian, as root
curl -sL https://deb.nodesource.com/setup_7.x | bash -
apt-get install -y nodejs

# Using RHEL or centos, as root
curl -sL https://rpm.nodesource.com/setup_7.x | bash -
yum install nodejs
```

Now, install the build tools:

```
npm -g install bower babel-cli babel-preset-es2015 babel-plugin-external-helpers less␣
↪coffee-script angular-gettext-cli angular-gettext-tools

# Ubuntu or Debian:
apt-get install gettext

# RHEL or CentOS
yum install gettext
```

3. Install ajenti-dev-multitool

pip install ajenti-dev-multitool

Your first plugin

Create a new plugin in the current directory:

```
ajenti-dev-multitool --new-plugin "Some plugin name"
```

Build resources:

```
cd some_plugin_name
ajenti-dev-multitool --build
```

And start it:

```
sudo ajenti-dev-multitool --run-dev
```

This will start Ajenti with the stock plugins plus the current one, and will rebuild plugin resources every time you reload Ajenti in browser.

Navigate to http://localhost:8000/. You should see new plugin in the sidebar.

What's inside

Each plugin package consists of Python modules, which contain `jadi.component` classes (*components*). Packages also may contain *static files, templates and JS and CSS code*, e.g.:

```
* some_plugin_name
  - plugin.yml # plugin description
  - __init__.py
  - other_python_module.py
  * resources
    * vendor
      - jquery-ui # Bower components
  * js
    - module.js # Angular.js code
    - ecmascript6-code.es
    - coffeescript-code.coffee
  * css
    - styles.css
    - styles.less
  * partials
    -- index.html
```

Where to go from here

Example plugins

Download plugins from here: https://github.com/ajenti/demo-plugins or clone this entire repository.

Prep work:

```
ajenti-dev-multitool --bower install
ajenti-dev-multitool --rebuild
```

Run:

```
ajenti-dev-multitool --run-dev
```

Hint: Changes in ES6, 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. Reloading the page with Ctrl-F5 (`Cache-Control: no-cache`) will unconditionally rebuild all resources

Ajenti Dev Multitool

```
sudo pip install ajenti-dev-multitool
```

`ajenti-dev-multitool` is a mini-utility to help you with common plugin development tasks.

`ajenti-dev-multitool` typically operates on all plugins found in current directory and below.

- `--run` will launch the globally installed Ajenti with plugins from the current directory. `--run-dev` will additionally enable developer mode.
- `--bower "<bower-command-with-args>"` will run a Bower command for each plugin having its own `bower.json` file. Example: `ajenti-dev-multitool --bower "install"`.
- `--build` updates the resource bundles. `--rebuild` will discard any previously built resources.
- `--setuppy "<setup.py-command-with-args>"` runs a setuptools command on the plugin package. A `setup.py` file is generated automatically. Example: `ajenti-dev-multitool --setuppy 'sdist upload --sign --identity "John Doe"'`

User Interface

Basics

Ajenti frontend is a AngularJS-based single-page rich web application.

Your plugins can extend it by adding new Angular components (services, controllers, directives) and routes (`ngRoute`).

Client-server communication is facilitated by AJAX requests to backend API (`$http`) and a Socket.IO connection (`socket` and `push` Angular services).

Client styling is based on a customized Twitter Bootstrap build.

Example

Basic UI example can be browsed and downloaded at https://github.com/ajenti/demo-plugins/tree/master/demo_2_ui

The basic UI plugin includes:

- an AngularJS module containing a route and a controller:
- an AngularJS view template (HTML)

Plugin resources

Basics

Plugin resource files are contained under `resources` directory nested in the plugin directory.

We encourage following structure:

```
* plugin
  * resources
    * css
      - styles.less
    * js
      - module.coffee
      - routing.coffee
      * controllers
        - some.controller.coffee
      * services
        - some.service.coffee
    * img
      - image.png
    * partials
      - view.html
```

CSS, JS and HTML resources must be listed in the `plugin.yml` file in order to be served to client. Example:

```
name: test
...
resources:
    - 'resources/vendor/jquery/dist/jquery.min.js'   # Bower component
    - 'resources/css/animations.less'                # Styles
    - 'resources/js/core/filters.coffee'             # JS
    - 'resources/partial/index.html'                 # HTML
    - 'ng:moduleName'                                # Special syntax for publishing␣
↪an AngularJS module.
```

Please note that the last item instructs Ajenti core to load the specified AngularJS module (`test`) from the plugin.

Bower components

Example can be browsed and downloaded at https://github.com/ajenti/demo-plugins/tree/master/demo_3_bower

Plugins can depend on Bower components. To use this feature, create a `bower.json` file in your plugin directory:

```
{
  "name": "plugin",
  "private": true,
  "dependencies": {
    "jquery": "~2.1.3"
  }
}
```

Components are installed into `<plugin>/resources/vendor` directory. To install/update the components, run `ajenti-dev-multitool --bower install`. You can also run `make bower` in the root of a complete Ajenti code tree to install Bower components in all plugins.

You can run other Bower commands with e.g. `ajenti-dev-multitool --bower "list --force --verbose"`.

Resource access

AngularJS templates are pre-loaded on the client. A template residing in `plugins/test/resources/dir/template.html` can be accessed with the following URL: `/test:resources/dir/template.html`.

Other resource files are available through HTTP at `/resources/<plugin_id>/resources/<path>`.

Resource compilation

When running in dev mode (`--dev`), Ajenti will invoke `ajenti-dev-multitool --build` on page reload. Force-reloading the page (Ctrl/Cmd-F5) will rebuild all resources in all plugins using `ajenti-dev-multitool --rebuild`

`ajenti-dev-multitool` will automatically compile CoffeeScript and LESS code, concatenate CSS and JS specified in `plugin.yml` and place built CSS and JS files in `plugin/resources/build`. Please note that `ajenti-dev-multitool` will only process files in the current directory and below.

Handling HTTP Requests

Example

Basic HTTP API example can be browsed and downloaded at https://github.com/ajenti/demo-plugins/tree/master/demo_4_http

Plugins can provide their own HTTP endpoints by extending the *aj.api.http.HttpPlugin* abstract class.

Example:

```
import time
from jadi import component

from aj.api.http import url, HttpPlugin

from aj.api.endpoint import endpoint, EndpointError, EndpointReturn


@component(HttpPlugin)
class Handler(HttpPlugin):
    def __init__(self, context):
        self.context = context

    @url(r'/api/demo4/calculate/(?P<operation>\w+)/(?P<a>\d+)/(?P<b>\d+)')
    @endpoint(api=True)
    def handle_api_calculate(self, http_context, operation=None, a=None, b=None):
        start_time = time.time()

        try:
            if operation == 'add':
                result = int(a) + int(b)
            elif operation == 'divide':
                result = int(a) / int(b)
            else:
                raise EndpointReturn(404)
        except ZeroDivisionError:
            raise EndpointError('Division by zero')

        return {
```

```
            'value': result,
            'time': time.time() - start_time
        }
```

`@endpoint(api=True)` mode provides automatic JSON encoding of the responses and error handling.

If you need lower-level access to the HTTP response, use `@endpoint(page=True)`:

```
@url(r'/api/test')
@endpoint(page=True)
def handle_api_calculate(self, http_context):
    http_context.add_header('Content-Type', '...')
    content = "Hello!"
    #return http_context.respond_not_found()
    #return http_context.respond_forbidden()
    #return http_context.file('/some/path')
    http_context.respond_ok()
    return content
```

See *aj.http.HttpContext* for the available `http_context` methods.

Dashboard Widgets

Example

Basic example of a dynamic and configurable widget can be browsed and downloaded at https://github.com/ajenti/demo-plugins/tree/master/demo_5_widget

Plugins can provide dashboard widgets by extending the *aj.plugins.dashboard.api.Widget* abstract class:

```
@component(Widget)
class RandomWidget(Widget):
    id = 'random'

    # display name
    name = 'Random'

    # template of the widget
    template = '/demo_5_widget:resources/partial/widget.html'

    # template of the configuration dialog
    config_template = '/demo_5_widget:resources/partial/widget.config.html'

    def __init__(self, context):
        Widget.__init__(self, context)

    def get_value(self, config):
        # generate value based on widget's config
        if 'bytes' not in config:
            return 'Not configured'
        return os.urandom(int(config['bytes'])).encode('hex')
```

There are some CSS classes available for the standard widget looks:

```
<div ng:controller="Demo5WidgetController">
    <div class="widget-header">
        Random
```

```
    </div>
    <div class="widget-value">
        {{value || 'Unknown'}}
    </div>
</div>
```

The templates should reference appropriate controllers:

```
angular.module('ajenti.demo5').controller 'Demo5WidgetController', ($scope) ->
    # $scope.widget is our widget descriptor here
    $scope.$on 'widget-update', ($event, id, data) ->
        if id != $scope.widget.id
            return
        $scope.value = data


angular.module('ajenti.demo5').controller 'Demo5WidgetConfigController', ($scope) ->
    # $scope.configuredWidget is our widget descriptor here
    # some defaults
    $scope.configuredWidget.config.bytes ?= 4
```

Initially, dashboard will create your widget with an empty ({}) config and show the configuration dialog you provided.

Dashboard will issue periodic requests to your *aj.plugins.dashboard.api.Widget* implementations. Your widget classes should not retain any state. If user creates multiple widgets of same type, a single instance will be created to service their requests.

Getting Started with Core Development

Attention: This article is only useful to the developers interested in developing the Ajenti core itself. For plugin/extension development, see *Getting started with plugin development*

Required knowledge

- Python 2.x
- async programming with gevent
- HTML
- CoffeeScript (with AngularJS)
- LESS

Prerequisites

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

- git
- bower, babel, babel-preset-es2015 and lessc (from NPM)

Debian/Ubuntu extras:

- python-dbus (ubuntu)

Setting up

Download the source:

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

Install the dependencies:

```
# Debian/Ubuntu
sudo apt-get install build-essential python-pip python-dev python-lxml libffi-dev␣
↪libssl-dev libjpeg-dev libpng-dev uuid-dev python-dbus``

# RHEL/CentOS
sudo yum install gcc python-devel python-pip libxslt-devel libxml2-devel libffi-devel␣
↪openssl-devel libjpeg-turbo-devel libpng-devel dbus-python

sudo pip install -r ajenti-core/requirements.txt
sudo pip install ajenti-dev-multitool

sudo npm install -g coffee-script less bower
```

Download and install Bower dependencies:

```
make bower
```

Ensure that resource compilation is set up correctly and works (optional):

```
make build
```

Launch Ajenti in dev mode:

```
make rundev
```

Navigate to http://localhost:8000/.

Hint: Changes in 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. Reloading the page with Ctrl-F5 (`Cache-Control: no-cache`) will unconditionally rebuild all resources

API: jadi

`jadi.`**`get_fqdn`**(*cls*)
Returns a fully-qualified name for the given class

`jadi.`**`interface`**(*cls*)
Marks the decorated class as an abstract interface.

Injects following classmethods:

`.all`(*context*)
Returns a list of instances of each component in the `context` implementing this `@interface`
Parameters **`context`** (*Context*) – context to look in
Returns list(`cls`)

`.any`(*context*)
Returns the first suitable instance implementing this `@interface` or raises *NoImplementationError* if none is available.

Parameters **context** (*Context*) – context to look in
Returns cls

.classes()
Returns a list of classes implementing this @interface
Returns list(class)

jadi.**component**(*iface*)
Marks the decorated class as a component implementing the given iface

Parameters **iface** (*interface()*) – the interface to implement

jadi.**service**(*cls*)
Marks the decorated class as a singleton service.

Injects following classmethods:

.get(*context*)
Returns a singleton instance of the class for given context
Parameters **context** (*Context*) – context to look in
Returns cls

class jadi.**Context**(*parent=None*)
An IoC container for *interface()* s, *service()* s and *component()* s

Parameters **parent** (*Context*) – a parent context

get_component(*cls*)

get_components(*cls*, *ignore_exceptions=False*)

get_service(*cls*)

exception jadi.**NoImplementationError**(*cls*)

API: aj

aj.**platform = 'debian'**
Current platform

aj.**platform_string = 'Ubuntu 16.04.2 LTS'**
Human-friendly platform name

aj.**platform_unmapped = 'ubuntu'**
Current platform without "Ubuntu is Debian"-like mapping

aj.**version = '2.1.18'**
Ajenti version

aj.**server = None**
Web server

aj.**debug = False**
Debug mode

aj.**init**()

aj.**exit**()

aj.**restart**()

API: aj.api.http

class aj.api.http.**BaseHttpHandler**
Base class for everything that can process HTTP requests

handle(*http_context*)
Should create a HTTP response in the given http_context and return the plain output

Parameters **http_context** (*aj.http.HttpContext*) – HTTP context

class aj.api.http.**HttpMiddleware**(*context*)

handle(*http_context*)

class aj.api.http.**HttpPlugin**(*context*)
A base interface for HTTP request handling:

```
@component
class HelloHttp(HttpPlugin):
    @url('/hello/(?P<name>.+)')
    def get_page(self, http_context, name=None):
        context.add_header('Content-Type', 'text/plain')
        context.respond_ok()
        return 'Hello, %s!' % name
```

handle(*http_context*)
Finds and executes the handler for given request context (handlers are methods decorated with *url()*)

Parameters **http_context** (*aj.http.HttpContext*) – HTTP context

Returns reponse data

class aj.api.http.**SocketEndpoint**(*context*)
Base interface for Socket.IO endpoints.

destroy()
Destroys endpoint, killing the running greenlets

on_connect(*message*)
Called on a successful client connection

on_disconnect(*message*)
Called on a client disconnect

on_message(*message*, **args*)
Called when a socket message arrives to this endpoint

plugin = None
arbitrary plugin ID for socket message routing

send(*data*, *plugin=None*)
Sends a message to the client.the

Parameters

- **data** – message object
- **plugin** (*str*) – routing ID (this endpoint's ID if not specified)

spawn(*target*, **args*, ***kwargs*)
Spawns a greenlet in this endpoint, which will be auto-killed when the client disconnects

Parameters **target** – target function

aj.api.http.**url**(*pattern*)
Exposes the decorated method of your *HttpPlugin* via HTTP

Parameters **pattern** (*str*) – URL regex (^ and $ are implicit)

Return type

function

Named capture groups will be fed to function as **kwargs

API: aj.api.endpoint

exception aj.api.endpoint.**EndpointError**(*inner*, *message=None*)
To be raised by endpoints when a foreseen error occurs. This exception doesn't cause a client-side crash dialog.

Parameters

- **inner** – inner exception
- **message** – message

exception aj.api.endpoint.**EndpointReturn**(*code*, *data=None*)
Raising EndpointReturn will return a custom HTTP code in the API endpoints.

Parameters

- **code** – HTTP code
- **data** – response data

aj.api.endpoint.**endpoint**(*page=False*, *api=False*, *auth=True*)
It's recommended to decorate all HTTP handling methods with @endpoint.

@endpoint(auth=True) will require authenticated session before giving control to the handler.

@endpoint(api=True) will wrap responses and exceptions into JSON, and will also provide special handling of EndpointsError

Parameters

- **auth** (*bool*) – requires authentication for this endpoint
- **page** (*bool*) – enables page mode
- **api** (*bool*) – enables API mode

API: aj.config

class aj.config.**BaseConfig**
A base class for config implementations. Your implementation must be able to save arbitrary mixture of dict, list, and scalar values.

data
currenly loaded config content

ensure_structure()

load()
Should load config content into *data*.

save()
Should save config content from *data*.

class aj.config.**UserConfig**(*context*)

get(*context*)

harden()

load()

save()

API: aj.core

aj.core.**run**(*config=None*, *plugin_providers=None*, *product_name=u'ajenti'*, *dev_mode=False*, *debug_mode=False*, *autologin=False*)
A global entry point for Ajenti.

Parameters

- **config** (*aj.config.BaseConfig*) – config file implementation instance to use
- **plugin_providers** (list(*aj.plugins.PluginProvider*)) – list of plugin providers to load plugins from
- **product_name** (*str*) – a product name to use
- **dev_mode** (*bool*) – enables dev mode (automatic resource recompilation)
- **debug_mode** (*bool*) – enables debug mode (verbose and extra logging)
- **autologin** (*bool*) – disables authentication and logs everyone in as the user running the panel. This is EXTREMELY INSECURE.

API: aj.entry

aj.entry.**handle_crash**(*exc*)

aj.entry.**start**(*daemonize=False*, *log_level=20*, ***kwargs*)
A wrapper for `run()` that optionally runs it in a forked daemon process.

Parameters **kwargs** – rest of arguments is forwarded to `run()`

API: aj.http

class aj.http.**HttpContext**(*env*, *start_response=None*)
Instance of *HttpContext* is passed to all HTTP handler methods

env
WSGI environment dict

path
Path segment of the URL

method
Request method

headers
List of HTTP response headers

body
Request body

response_ready
Indicates whether a HTTP response has already been submitted in this context

query
HTTP query parameters

add_header(*key*, *value*)
Adds a given HTTP header to the response

Parameters

- **key** (*str*) – header name
- **value** (*str*) – header value

classmethod deserialize(*data*)

dump_env()

fallthrough(*handler*)
Executes a `handler` in this context

Returns handler-supplied output

file(*path*, *stream=False*, *inline=False*, *name=None*)
Returns a GZip compressed response with content of file located in `path` and correct headers

get_cleaned_env()

gzip(*content*, *compression=6*)
Returns a GZip compressed response with given `content` and correct headers

Parameters **compression** (*int*) – compression level from 0 to 9

Return type str

json_body()

redirect(*location*)
Returns a `HTTP 302 Found` redirect response with given `location`

remove_header(*key*)
Removed a given HTTP header from the response

Parameters **key** (*str*) – header name

respond(*status*)
Creates a response with given HTTP status line

respond_forbidden()
Returns a HTTP `403 Forbidden` response

respond_not_found()
Returns a `HTTP 404 Not Found` response

respond_ok()
Creates a `HTTP 200 OK` response

respond_server_error()
Returns a HTTP `500 Server Error` response

run_response()
Finalizes the response and runs WSGI's `start_response()`.

serialize()

class aj.http.**HttpMiddlewareAggregator**(*stack*)
Stacks multiple HTTP handlers together in a middleware fashion.

Parameters **stack** (list(*aj.api.http.BaseHttpHandler*)) – handler list

handle(*context*)

class aj.http.**HttpRoot**(*handler*)
A root WSGI middleware object that creates the *HttpContext* and dispatches it to an HTTP handler.

Parameters **handler** (*aj.api.http.BaseHttpHandler*) – next middleware handler

dispatch(*env*, *start_response*)
Dispatches the WSGI request

API: aj.plugins

class aj.plugins.**PluginProvider**
A base class for plugin locator

provide()
Should return a list of found plugin paths

Returns list(str)

class aj.plugins.**DirectoryPluginProvider**(*path*)
A plugin provider that looks up plugins in a given directory.

Parameters **path** – directory to look for plugins in

provide()

class aj.plugins.**PythonPathPluginProvider**
A plugin provider that looks up plugins on `$PYTHONPATH`

provide()

exception aj.plugins.**PluginLoadError**

exception aj.plugins.**PluginCrashed**(*exception*)

describe()

class aj.plugins.**Dependency**

exception Unsatisfied

describe()

reason()

Dependency.**build_exception**()

Dependency.**check**()

Dependency.**value**

class aj.plugins.**ModuleDependency**(*module_name=None*)

exception Unsatisfied

reason()

ModuleDependency.**description = 'Python module'**

ModuleDependency.**is_satisfied**()

class aj.plugins.**PluginDependency**(*plugin_name=None*)

exception Unsatisfied

reason()

PluginDependency.**description = 'Plugin'**

PluginDependency.**is_satisfied**()

class aj.plugins.**OptionalPluginDependency**(*plugin_name=None*)

exception Unsatisfied

reason()

OptionalPluginDependency.**description = 'Plugin'**

OptionalPluginDependency.**is_satisfied**()

class aj.plugins.**BinaryDependency**(*binary_name=None*)

exception Unsatisfied

reason()

BinaryDependency.**description = 'Application binary'**

BinaryDependency.**is_satisfied**()

class aj.plugins.**FileDependency**(*file_name=None*)

exception Unsatisfied

reason()

FileDependency.**description = 'File'**

FileDependency.**is_satisfied**()

class aj.plugins.**PluginManager**(*context*)
Handles plugin loading and unloading

get(*context*)

get_content_path(*name*, *path*)

get_crash(*name*)

get_loaded_plugins_list()

load_all_from(*providers*)
Loads all plugins provided by given providers.

Parameters **providers** (list(*PluginProvider*)) –

Angular: ajenti.core

This Angular module contains core components of Ajenti frontend.

Services

class config()

data
Config file content object

load()
Gets complete configuration data of the backend

Returns promise

save()
Updates and saves configuration data

Returns promise

getUserConfig()
Gets per-user configuration data of the backend

Returns promise → per-user Ajenti config object

setUserConfig(*config*)
Updates and saves per-user configuration data

Arguments

- **config** (*object*) – updated configuration data from `getUserConfig()`

Returns promise

class core()

pageReload()
Reloads the current URL

restart()
Restarts the Ajenti process

class hotkeys()
Captures shortcut key events

ENTER, ESC
Respective key codes

on(*scope*, *handler*, *mode='keydown'*)
Registers a hotkey handler in the provided `scope`

Arguments

- **scope** (*$scope*) – `$scope` to install handler into
- **handler** (*function(keyCode,rawEvent)*) – handler function. If the function returns a truthy value, event is cancelled and other handlers aren't notified.
- **mode** (*string*) – one of `keydown`, `keypress` or `keyup`.

class identity()

Provides info on the authentication status and user/machine identity

user

Name of the logged in user

effective

Effective UID of the server process

machine.**name**

User-provided name of the machine

isSuperuser

Whether current user is a superuser or not

auth(*username*, *password*, *mode*)

Attempts to authenticate current session as `username:password` with a `mode` of `normal` or `sudo`

login()

Redirects user to a login dialog

logout()

Deauthenticates current session

elevate()

Redirects user to a sudo elevation dialog

class messagebox()

Provides interface to modal messagebox engine

show(*options*)

Opens a new messagebox.

Arguments

- **options** (*object*) –
- **options.title** (*string*) –
- **options.text** (*string*) –
- **options.positive** (*string*) – positive action button text. Clicking it will resolve the returned promise.
- **options.negative** (*string*) – negative action button text. Clicking it will reject the returned promise.
- **options.template** (*string*) – (optional) custom body template
- **options.scrollable** (*boolean*) – whether message body is scrollable
- **options.progress** (*boolean*) – whether to display an indeterminate progress indicator in the message

Returns a Promise-like object with an additional `close()` method.

class notify()

info (*title*, *text*)

success (*title*, *text*)

warning (*title*, *text*)

error (*title*, *text*)
Shows an appropriately styled notification

custom (*style*, *title*, *text*, *url*)
Shows a clickable notification leading to `url`.

class **pageTitle** ()
Alters page `<title>` and global heading.

set (*text*)
Sets title text

set (*expression*, *scope*)
Sets an title expression to be watched. Example:

```
$scope.getTitle = (page) -> someService.getPageTitle(page)
$scope.page = ...

pageTitle.set("getTitle(page)", $scope)
```

class **push** ()
Processes incoming push messages (see *aj.plugins.core.api.push*). This service has no public methods.

This service broadcasts events that can be received as:

```
$scope.$on 'push:pluginname', (message) ->
    processMessage(message)...
```

class **tasks** ()
An interface to the tasks engine (see *aj.plugins.core.api.tasks*).

tasks
A list of task descriptors for the currently running tasks. Updated automatically.

start (*cls*, *args*, *kwargs*)
Starts a server-side task.

Arguments

- **cls** (*string*) – full task class name (`aj.plugins.pluginname....`)
- **args** (*array*) – task arguments
- **kwargs** (*object*) – task keyword arguments

Returns a promise, resolved once the task actually starts

Directives

autofocus ()
Automatically focuses the input. Example:

```
<input type="text" autofocus ng:model="..." />
```

checkbox()
Renders a checkbox. Example:

```
<span checkbox ng:model="..." text="Enable something"></span>
```

dialog()
A modal dialog

Example:

```
<dialog ng:show="showDialog">
    <div class="modal-header">
        <h4>
            Heading
        </h4>
    </div>
    <div class="modal-body scrollable">
        ...
    </div>
    <div class="modal-footer">
        <a ng:click="..." class="btn btn-default btn-flat">
            Do something
        </a>
    </div>
</dialog>
```

Arguments

- **ngShow** (*expression*) –
- **dialogClass** (*string*) –

floating-toolbar()
A toolbar pinned to the bottom edge. Example:

```
<div class="floating-toolbar-padder"></div>

<floating-toolbar>
    <a ng:click="..." class="btn btn-default btn-flat">
        Do something useful
    </a>
</floating-toolbar>

<!-- accented toolbar for selection actions -->

<floating-toolbar class="accented" ng:show="haveSelectedItems">
    Some action buttons here
</floating-toolbar>
```

ng-enter()
Action handler for Enter key in inputs. Example:

```
<input type="text" ng:enter="commitStuff()" ng:model="..." />
```

progress-spinner()

root-access()
Blocks its inner content if the current user is not a superuser.

smart-progress()
An improved version of ui-bootstrap's progressbar

Arguments

- **animate** (*boolean*) –
- **value** (*float*) –
- **max** (*float*) –
- **text** (*string*) –
- **maxText** (*string*) –

Filters

bytesFilter(*value*, *precision*)

Arguments

- **value** (*int*) – number of bytes
- **precision** (*int*) – number of fractional digits in the output

Returns string, e.g.: `123.45 KB`

ordinalFilter(*value*)

Arguments

- **value** (*int*) –

Returns string, e.g.: `121st`

pageFilter(*list*, *page*, *pageSize*)
Provides a page-based view on an array

Arguments

- **list** (*array*) – input data
- **page** (*int*) – 1-based page index
- **pageSize** (*int*) – page size

Returns array

Angular: ajenti.ace

ACE code editor integration

Directives

ace-editor()

Arguments

- **ngModel** (*binding*) –
- **aceOptions** (*object*) – (optional) options for ace.setOptions()

Angular: ajenti.augeas

Services

class augeas()

get(*endpoint*)
Reads an Augeas tree from server side.

Returns promise → AugeasConfig

set(*endpoint*, *config*)
Overwrites an Augeas tree on the server side.

Returns promise

class AugeasNode()

name

value

parent

children

fullPath()

class AugeasConfig()
This is a JS doppelganger of normal Augeas API. In particular, it doesn't support advanced XPath syntax, and operates with regular expressions instead.

get(*path*)

Returns `AugeasNode`

set(*path*, *value*)

model(*path*)

Returns a getter/setter function suitable for use as a `ngModel`

insert(*path*, *value*, *index*)

remove(*path*)

match(*path*)

Returns `Array(string)`

matchNodes(*path*)

Returns `Array(AugeasNode)`

Angular: ajenti.filesystem

Services

class filesystem()

read(*path*)

Returns promise → content of `path`

write (*path*, *content*)

Returns promise

list (*path*)

Returns promise → array

stat (*path*)

Returns promise → object

chmod (*path*, *mode*)

Arguments

- **mode** (*int*) – numeric POSIX file mode

Returns promise

createFile (*path*, *mode*)

Arguments

- **mode** (*int*) – numeric POSIX file mode

Returns promise

createDirectory (*path*, *mode*)

Arguments

- **mode** (*int*) – numeric POSIX file mode

Returns promise

downloadBlob (*content*, *mime*, *name*)

Launches a browser-side file download

Arguments

- **content** (*string*) – Raw file content
- **mime** (*string*) – MIME type used
- **name** (*string*) – Default file name for saving

Returns promise

Directives

file-dialog ()

File open/save dialog. Example:

```
<file-dialog
    mode="open"
    ng:show="openDialogVisible"
    on-select="open(item.path)"
    on-cancel="openDialogVisible = false">
</file-dialog>

<file-dialog
    mode="save"
    ng:show="saveDialogVisible"
```

```
    on-select="saveAs(path)"
    on-cancel="saveDialogVisible = false"
    name="saveAsName">
</file-dialog>
```

Arguments

- **ngShow** (*expression*) –
- **onSelect** (*expression(item)*) – called after opening or saving a file. `item` is an object with a `path` property.
- **onCancel** (*expression*) – (optional) handler for the cancel button
- **mode** (*string*) – one of `open`, `save`
- **name** (*binding*) – (optional) name for the saved file
- **path** (*binding*) – (optional) current

path-selector()

An input with a file selection dialog:

```
<path-selector ng:model="filePath"></path-selector>
```

Angular: ajenti.passwd

Services

class passwd()

list()

Returns promise → array of the users registered in the system

Angular: ajenti.services

Services

class services()

getManagers()

Returns promise → array of the available service managers

getServices(*managerId*)

Returns promise → array of the available services in the `ServiceManager`

getService(*managerId*, *serviceId*)

Returns promise → object, gets a single service from the manager

runOperation(*managerId*, *serviceId*, *operation*)

Arguments

- **operation** (*string*) – typically `start`, `stop`, `restart`, `reload`; depends on the service manager

Returns promise

Angular: ajenti.terminal

Services

class **terminals**()

list()

Returns promise → array of opened terminal descriptors

kill(*terminalId*)
Kills a running terminal process

Returns promise

create(*options*)
Creates a new terminal

Arguments

- **options.command** (*string*) –
- **options.autoclose** (*boolean*) –

Returns promise → new terminal ID

full(*terminalId*)

Returns promise → full content of the requested terminal

Plugin: aj.plugins.core.api.push

class `aj.plugins.core.api.push.`**Push**(*context*)
A service providing push messages to the client.

get(*context*)

push(*plugin*, *msg*)
Sends a push message to the client.

Parameters

- **plugin** – routing ID
- **msg** – message

register()

Plugin: aj.plugins.core.api.sidebar

class `aj.plugins.core.api.sidebar.`**Sidebar**(*context*)

build()
Returns a complete tree of sidebar items.

Returns dict

get(*context*)

class aj.plugins.core.api.sidebar.**SidebarItemProvider**(*context*)
Interface for providing sidebar items.

provide()
Should return a list of sidebar items, each in the following format:

```
{
    'id': 'optional-id',
    'attach': 'category:general', # id of the attachment point or None for␣
↪top level
    'name': 'Dashboard',
    'icon': 'bar-chart',
    'url': '/view/dashboard',
    'children': [
        ...
    ]
}
```

Returns list(dict)

Plugin: aj.plugins.core.api.tasks

class aj.plugins.core.api.tasks.**Task**(*context*, **args*, ***kwargs*)
Tasks are one-off child processes with progress reporting. This is a base abstract class.

abort()

name = None
Display name

push(*plugin*, *message*)
An interface to *aj.plugins.core.api.push.Push* usable from inside the task's process

report_progress(*message=None*, *done=None*, *total=None*)
Updates the task's process info.

Parameters

- **message** – text message
- **done** – number of processed items
- **total** – total number of items

run()
Override this with your task's logic.

send_log_event(*method*, *message*, **args*, ***kwargs*)

start()
Starts the task's process

class aj.plugins.core.api.tasks.**TasksService**(*context*)

abort(*_id*)

format_tasks()

get(*context*)

notify(*message=None*)

remove(*_id*)

send_update()

start(*task*)

Plugin: aj.plugins.augeas.api

class aj.plugins.augeas.api.**Augeas**(*modules=[]*, *loadpath=None*)
A smarter and faster wrapper around augeas.Augeas.augeas

For faster startup, no modules and lenses are preloaded:

```
aug = Augeas(modules=[{
    'name': 'Interfaces',  # module name
    'lens': 'Interfaces.lns',  # lens name
    'incl': [  # included files list
        self.path,
        self.path + '.d/*',
    ]
}])
```

Don't forget to call load() afterwards.

dump(*path*)
Dumps contents under *path* to stdout.

get(*path*)

match(*path*)

raise_error()
Extracts error information from Augeas tree and raises *AugeasError*

save()

set(*path*, *value*)

setd(*path*, *value*, *default=None*)
Sets *path* to *value*, or removes *path* if *value* == *default*

class aj.plugins.augeas.api.**AugeasEndpoint**(*context*)
Implement this to provide Augeas trees to the frontend.

get_augeas()
Should return a ready-to-use *Augeas*

get_root_path()
Should return an Augeas path of the root node to be provided to the frontend.

id = None

exception aj.plugins.augeas.api.**AugeasError**(*aug*)

Plugin: aj.plugins.dashboard.api

class aj.plugins.dashboard.api.**Widget**(*context*)
Base interface for dashboard widgets.

config_template = None
Configuration dialog template URL

get_value(*config*)
Override this to return the widget value for the given config dict.

id = None

name = None
Display name

template = None
Angular view template URL

Plugin: aj.plugins.services.api

class aj.plugins.services.api.**Service**(*manager*)

class aj.plugins.services.api.**ServiceManager**

get_service(*_id*)

id = None

list()

name = None

restart(*_id*)

start(*_id*)

stop(*_id*)

exception aj.plugins.services.api.**ServiceOperationError**(*inner*)

Python Module Index

a

j

Index

S

T

U

V

W