circuits Documentation

Release 2.0.1

James Mills

March 03, 2013

CONTENTS

Release 2.0.1

Date March 03, 2013

CHAPTER

ONE

CONTENTS

1.1 Getting Started

1.1.1 Quick Start Guide

The easiest way to download and install circuits is to use the pip command:

```
$ pip install circuits
```

Now that you have successfully downloaded and installed circuits, let's test that circuits is properly installed and working.

First, let's check the installed version:

```
>>> import circuits
>>> print circuits.__version__
1.3
```

Try some of the examples in the examples/ directory shipped with the distribution or check out some *Applications using circuits*

Have fun :)

1.1.2 Downloading

Latest Stable Release

The latest stable releases can be downloaded from the Downloads page.

Latest Development Source Code

We use Mercurial for source control and code sharing.

The latest development branch can be cloned using the following command:

```
$ hg clone http://bitbucket.org/prologic/circuits/
```

For further instructions on how to use Mercurial, please refer to the Mercurial Book.

1.1.3 Installing

If you have not installed circuits via the the setuptools easy_install tool, then the following installation instructions will apply to you. Either you've downloaded a source package or cloned the development repository.

Installing from a Source Package

```
$ python setup.py install
```

For other installation options see:

```
$ python setup.py --help install
```

Installing from the Development Repository

If you have cloned the development repository, it is recommended that you use setuptools and use the following command:

```
$ python setup.py build develop
```

NB: The "build" command is required when installing from the development repository (*build creates the version file which is built dynamically*).

This will allow you to regularly update your copy of the circuits development repository by simply performing the following in the circuits working directory:

```
$ hg pull -u
```

NB: You do not need to reinstall if you have installed with setuptools via the circuits repository and used setuptools to install in "develop" mode.

1.1.4 Requirements and Dependencies

- circuits has no **required** dependencies beyond the Python Standard Library.
- Python: >= 2.6

Other Optional Dependencies

These dependencies are not strictly required and only add additional features such as the option for a routes dispatcher for circuits.web and rendering of component graphs for your application.

- pydot – For rendering component graphs of an application.

1.2 Tutorial

1.2.1 Overview

Welcome to the circuits tutorial. This 5-minute tutorial will guide you through the basic concepts of circuits. The goal is to introduce new concepts incrementally with walk-through examples that you can try out! By the time you've finished, you should have a good basic understanding of circuits, how it feels and where to go from there.

1.2.2 The Component

First up, let's show how you can use the `Component` and run it in a very simple application.

```
#!/usr/bin/env python

from circuits import Component

Component().run()
```

Download 001.py

Okay so that's pretty boring as it doesn't do very much! But that's okay... Read on!

Let's try to create our own custom Component called `MyComponent`.

```
#!/usr/bin/env python

from circuits import Component

class MyComponent(Component):
    """My Component"""

MyComponent().run()
```

Download 002.py

Okay, so this still isn't very useful! But at least we can create components with the behavior we want.

Let's move on to something more interesting...

1.2.3 Event Handlers

Let's now extend our little example to say "Hello World!" when its started.

```
#!/usr/bin/env python

from circuits import Component

class MyComponent(Component):

    def started(self, *args):
        print("Hello World!")

MyComponent().run()
```

Download 003.py

Here we've created a simple **Event Handler** that listens for events on the "started" channel. Methods defined in a Component are converted into Event Handlers.

Running this we get:

```
Hello World!
```

Alright! We have something slightly more useful! Whoohoo it says hello!

Note: Press ^C (*CTRL* + *C*) to exit.

1.2.4 Registering Components

So now that we've learned how to use a Component, create a custom Component and create simple Event Handlers, let's try something a bit more complex by creating a complex component made up of two simpler ones.

Let's create two components:

- Bob
- Fred

```
#!/usr/bin/env python

from circuits import Component

class Bob(Component):

    def started(self, *args):
        print("Hello I'm Bob!")

class Fred(Component):

    def started(self, *args):
        print("Hello I'm Fred!")

(Bob() + Fred()).run()
```

Download 004.py

Notice the way we register the two components Bob and Fred together ? Don't worry if this doesn't make sense right now. Think of it as putting two components together and plugging them into a circuits board.

Running this example produces the following result:

```
Hello I'm Bob!
Hello I'm Fred!
```

Cool! We have two components that each do something and print a simple message on the screen!

1.2.5 Complex Components

Now, what if we wanted to create a Complex Component ? Let's say we wanted to create a new Component made up of two other smaller components ?

We can do this by simply registering components to a Complex Component during initialization.

```
#!/usr/bin/env python

from circuits import Component
from circuits.tools import graph

class Pound(Component):

    def __init__(self):
        super(Pound, self).__init__()

        self.bob = Bob().register(self)
        self.fred = Fred().register(self)

    def started(self, *args):
```

```
15          print(graph(self.root))
16
17  class Bob(Component):
18
19      def started(self, *args):
20          print("Hello I'm Bob!")
21
22  class Fred(Component):
23
24      def started(self, *args):
25          print("Hello I'm Fred!")
26
27  Pound().run()
```

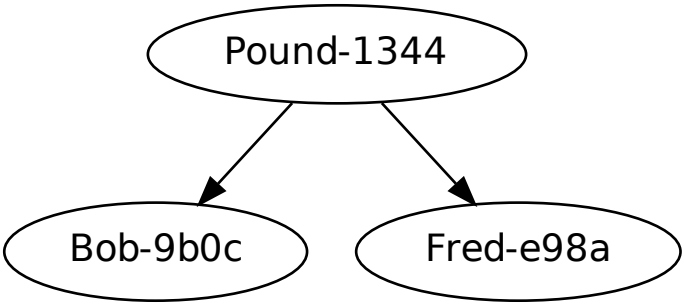
Download 005.py

So now `Pound` is a Component that consists of two other components registered to it: `Bob` and `Fred`

The output of this is identical to the previous:

```
* <Pound/* 3391:MainThread (queued=0, channels=1, handlers=3) [R]>
 * <Bob/* 3391:MainThread (queued=0, channels=1, handlers=1) [S]>
 * <Fred/* 3391:MainThread (queued=0, channels=1, handlers=1) [S]>
Hello I'm Bob!
Hello I'm Fred!
```

The only difference is that `Bob` and `Fred` are now part of a more Complex Component called `Pound`. This can be illustrated by the following diagram:



Note: The extra lines in the above output are an ASCII representation of the above graph (*produced by pydot + graphviz*).

Cool :-)

1.2.6 Component Inheritence

Since circuits is a framework written for the Python Programming Language it naturally inherits properties of Object Orientated Programming (OOP) – such as inheritence.

So let's take our `Bob` and `Fred` components and create a Base Component called `Dog` and modify our two dogs (`Bob` and `Fred`) to subclass this.

```
#!/usr/bin/env python

from circuits import Component, Event


class Woof(Event):
    """Woof Event"""


class Pound(Component):

    def __init__(self):
        super(Pound, self).__init__()

        self.bob = Bob().register(self)
        self.fred = Fred().register(self)

    def started(self, *args):
        self.fire(Woof())


class Dog(Component):

    def woof(self):
        print("Woof! I'm %s!" % self.name)


class Bob(Dog):
    """Bob"""


class Fred(Dog):
    """Fred"""

Pound().run()
```

Download 006.py

Now let's try to run this and see what happens:

```
Woof! I'm Bob!
Woof! I'm Fred!
```

So both dogs barked~ Hmmm

1.2.7 Component Channels

What if we only want one of our dogs to bark ? How do we do this without causing the other one to bark as well ?

Easy! Use a separate `channel` like so:

```
#!/usr/bin/env python

from circuits import Component, Event


```

```
6  class Woof(Event):
7      """Woof Event"""
8
9
10 class Pound(Component):
11
12     def __init__(self):
13         super(Pound, self).__init__()
14
15         self.bob = Bob().register(self)
16         self.fred = Fred().register(self)
17
18     def started(self, *args):
19         self.fire(Woof(), self.bob)
20
21
22 class Dog(Component):
23
24     def woof(self):
25         print("Woof! I'm %s!" % self.name)
26
27
28 class Bob(Dog):
29     """Bob"""
30
31     channel = "bob"
32
33
34 class Fred(Dog):
35     """Fred"""
36
37     channel = "fred"
38
39 Pound().run()
```

Download 007.py

Note: Events can be fired with either the `.fire(...)` or `.fireEvent(...)` method.

If you run this, you'll get:

```
Woof! I'm Bob!
```

1.2.8 Event Objects

So far in our tutorial we have been defining an Event Handler for a builtin Event called `Started` (*which incidentally gets fired on a channel called "started"*). What if we wanted to define our own Event Handlers and our own Events ? You've already seen how easy it is to create a new Event Handler by simply defining a normal Python method on a Component.

Defining your own Events helps with documentation and testing and makes things a little easier.

Example:

```
class MyEvent(Event):
   """MyEvent"""
```

So here's our example where we'll define a new Event called `Bark` and make our `Dog` fire a `Bark` event when our application starts up.

```
#!/usr/bin/env python

from circuits import Component, Event

class Bark(Event):
    """Bark Event"""

class Pound(Component):

    def __init__(self):
        super(Pound, self).__init__()

        self.bob = Bob().register(self)
        self.fred = Fred().register(self)

class Dog(Component):

    def started(self, *args):
        self.fire(Bark())

    def bark(self):
        print("Woof! I'm %s!" % self.name)

class Bob(Dog):
    """Bob"""

    channel = "bob"

class Fred(Dog):
    """Fred"""

    channel = "fred"

Pound().run()
```

Download 008.py

If you run this, you'll get:

```
Woof! I'm Bob!
Woof! I'm Fred!
```

1.2.9 The Debugger

Lastly...

Asynchronous programming has many advantages but can be a little harder to write and follow. A silently caught exception in an Event Handler, or an Event that never gets fired, or any number of other weird things can cause your application to fail and leave you scratching your head.

Fortunately circuits comes with a `Debugger` Component to help you keep track of what's going on in your application, and allows you to tell what your application is doing.

Let's say that we defined out `bark` Event Handler in our `Dog` Component as follows:

```
def bark(self):
   print("Woof! I'm %s!" % name)
```

Now clearly there is no such variable as `name` in the local scope.

For reference here's the entire example...

```
1  #!/usr/bin/env python
2
3  from circuits import Component, Event
4
5  class Bark(Event):
6      """Bark Event"""
7
8  class Pound(Component):
9
10     def __init__(self):
11         super(Pound, self).__init__()
12
13         self.bob = Bob().register(self)
14         self.fred = Fred().register(self)
15
16 class Dog(Component):
17
18     def started(self, *args):
19         self.fire(Bark())
20
21     def bark(self):
22         print("Woof! I'm %s!" % name)
23
24 class Bob(Dog):
25     """Bob"""
26
27     channel = "bob"
28
29 class Fred(Dog):
30     """Fred"""
31
32     channel = "fred"
33
34 Pound().run()
```

`Download 009.py`

If you run this, you'll get:

That's right! You get nothing! Why ? Well in circuits any error or exception that occurs in a running application is automatically caught and dealt with in a way that lets your application "keep on going". Crashing is unwanted behavior in a system so we expect to be able to recover from horrible situations.

SO what do we do ? Well that's easy. circuits come with a `Debugger` that lets you log all events as well as all errors so you can quickly and easily discover which Event is causing a problem and which Event Handler to look at.

If you change Line 34 of our example...

From:

```
Pound().run()
```

```
from circuits import Debugger

(Pound() + Debugger()).run()
```

Then run this, you'll get the following:

```
<Registered[bob:registered] [<Bob/bob 3191:MainThread (queued=0, channels=2, handlers=2) [S]>, <Pound
<Registered[fred:registered] [<Fred/fred 3191:MainThread (queued=0, channels=2, handlers=2) [S]>, <Po
<Registered[*:registered] [<Debugger/* 3191:MainThread (queued=0, channels=1, handlers=1) [S]>, <Pour
<Started[*:started] [<Pound/* 3191:MainThread (queued=0, channels=5, handlers=5) [R]>, None] {}>
<Bark[bob:bark] [] {}>
<Bark[fred:bark] [] {}>
<Error[*:exception] [<type 'exceptions.NameError'>, NameError("global name 'name' is not defined",),
ERROR <listener on ('bark',) {target='bob', priority=0.0}> (<type 'exceptions.NameError'>): global na
  File "/home/prologic/work/circuits/circuits/core/manager.py", line 459, in __handleEvent
 retval = handler(*eargs, **ekwargs)
  File "source/tutorial/009.py", line 22, in bark
    print("Woof! I'm %s!" % name)

<Error[*:exception] [<type 'exceptions.NameError'>, NameError("global name 'name' is not defined",),
ERROR <listener on ('bark',) {target='fred', priority=0.0}> (<type 'exceptions.NameError'>): global n
  File "/home/prologic/work/circuits/circuits/core/manager.py", line 459, in __handleEvent
    retval = handler(*eargs, **ekwargs)
  File "source/tutorial/009.py", line 22, in bark
    print("Woof! I'm %s!" % name)

^C<Signal[*:signal] [2, <frame object at 0x808e8ec>] {}>
<Stopped[*:stopped] [<Pound/* 3191:MainThread (queued=0, channels=5, handlers=5) [S]>] {}>
<Stopped[*:stopped] [<Pound/* 3191:MainThread (queued=0, channels=5, handlers=5) [S]>] {}>
```

You'll notice whereas there was no output before there is now a pretty detailed output with the `Debugger` added to the application. Looking through the output, we find that the application does indeed start correctly, but when we fire our `Bark` Event it coughs up two exceptions, one for each of our dogs (`Bob` and `Fred`).

From the error we can tell where the error is and roughly where to look in the code.

Note: You'll notice many other events that are displayed in the above output. These are all default events that circuits has builtin which your application can respond to. Each builtin Event has a special meaning with relation to the state of the application at that point.

See: `circuits.core.events` for detailed documentation regarding these events.

The correct code for the `bark` Event Handler should be:

```
def bark(self):
    print("Woof! I'm %s!" % self.name)
```

Running again with our correction results in the expected output:

```
Woof! I'm Bob!
Woof! I'm Fred!
```

That's it folks!

Hopefully this gives you a feel of what circuits is all about and a easy tutorial on some of the basic concepts. As you're no doubt itching to get started on your next circuits project, here's some recommended reading:

- *Frequently Asked Questions*
- *HowTos*

- *API Reference*

1.3 The circuits Framework

1.3.1 Components

The architectural concept of circuits is to encapsulate system functionality into discrete manageable and reusable units, called *Components*, that interact by sending and handling events that flow throughout the system.

Technically, a circuits *Component* is a Python class that inherits (directly or indirectly) from `BaseComponent`.

Components can be sub-classed like any other normal Python class, however components can also be composed of other components and it is natural to do so. These are called *Complex Components*. An example of a Complex Component within the circuits library is the `circuits.web.servers.Server` Component which is comprised of:

- `circuits.net.sockets.TCPServer`
- `circuits.web.servers.BaseServer`
- `circuits.web.http.HTTP`
- `circuits.web.dispatchers.dispatcher.Dispatcher`

Note that there is no class or other technical means to mark a component as a complex component. Rather, all component instances in a circuits based application belong to some component tree (there may be several), with Complex Components being a subtree within that structure.

A Component is attached to the tree by registering with the parent and detached by un-registering itself (methods `register()` and `unregister()` of `BaseComponent`).

1.3.2 Events

Basic usage

Events are objects that are fired by the circuits framework implicitly (like the `Started` event used in the tutorial) or explicitly by components while handling some other event. Once fired, events are dispatched to the components that are interested in these events, i.e. that have registered themselves as handlers for these events.

Events are usually fired on one or more channels, allowing components to gather in "interest groups". This is especially useful if you want to reuse basic components such as a TCP server. A TCP server component fires a `Read` event for every package of data that it receives. If we hadn't the channels, it would be very difficult to separate the data from two different TCP connections. But using the channels, we can put one TCP server and all components interested in its events on one channel, and another TCP server and the components interested in this other TCP server's events on another channel. Components are associated with a channel by setting their `channel` attribute (see API description for `Component`).

Besides having a name, events carry additional arbitrary information. This information is passed as arguments or keyword arguments to the constructor. It is then delivered to the handler function that must have exactly the same number of arguments and keyword arguments. Of course, as is usual in Python, you can also pass additional information by setting attributes of the event object, though this usage pattern is discouraged for events.

Events as result collectors

Apart from delivering information to handlers, event objects may also collect information. If a handler returns something that is not `None`, it is stored in the event's `value` attribute. If a second (or any subsequent) handler invocation also returns a value, the values are stored as a list. Note that the value attribute is of type `Value` and you must access its property `value` to access the data stored (`collected_information = event.value.value`).

The collected information can be accessed by handlers in order to find out about any return values from the previously invoked handlers. More useful though, is the possibility to access the information after all handlers have been invoked. After all handlers have run successfully (i.e. no handler has thrown an error) circuits may generate an event that indicates the successful handling. This event has the name of the event just handled with "Success" appended. So if the event is called `Identify` then the success event is called `IdentifySuccess`. Success events aren't delivered by default. If you want successful handling to be indicated for an event, you have to set the optional attribute `success` of this event to `True`.

The handler for a success event must be defined with two arguments. When invoked, the first argument is the event just having been handled successfully and the second argument is (as a convenience) what has been collected in `event.value.value` (note that the first argument may not be called `event`, for an explanation of this restriction as well as for an explanation why the method is called `identify_success` see the section on handlers).

```
#!/usr/bin/env python

from circuits import Component, Event
from circuits.core.debugger import Debugger

class Identify(Event):
    """Identify Event"""
    success = True

class Pound(Component):

    def __init__(self):
        super(Pound, self).__init__()

        Debugger().register(self)
        Bob().register(self)
        Fred().register(self)

    def started(self, *args):
        self.fire(Identify())

    def identify_success(self, evt, result):
        if not isinstance(result, list):
            result = [result]
        print "In pound:"
        for name in result:
            print name

class Dog(Component):

    def identify(self):
        return self.__class__.__name__

class Bob(Dog):
    """Bob"""

class Fred(Dog):
    """Fred"""
```

```
39
40 Pound().run()
```

Download handler_returns.py

Advanced usage

Sometimes it may be necessary to take some action when all state changes triggered by an event are in effect. In this case it is not sufficient to wait for the completion of all handlers for this particular event. Rather, we also have to wait until all events that have been fired by those handlers have been processed (and again wait for the events fired by those events' handlers, and so on). To support this scenario, circuits can fire a `Complete` event. The usage is similar to the previously described success event. Details can be found in the API description of `circuits.core.events.Event`.

1.3.3 Handlers

Explicit handler definition

Handlers are methods of components that are invoked when a matching event is dispatched. The question arises how methods are made known as handlers to the circuits framework.

The ability to define methods as handlers is already provided for in `Component`'s base class, the `BaseComponent`. Any class that inherits from `BaseComponent` can advertise a method as a handler using the handler annotation.

```
1  #!/usr/bin/env python
2
3  from circuits.core.debugger import Debugger
4  from circuits.core.components import BaseComponent
5  from circuits.core.handlers import handler
6
7  class MyComponent(BaseComponent):
8
9      def __init__(self):
10         super(MyComponent, self).__init__()
11
12         Debugger().register(self)
13
14     @handler("started", channel="*")
15     def _on_started(self, component):
16         print "Start event detected"
17
18 MyComponent().run()
```

Download handler_annotation.py

The handler annotation in line 14 makes the method `_on_started` known to circuits as a handler for the event `Started`. Event names used to define handlers are the uncameled class names of the event. An event with a class name `MySpecialEvent` becomes "`my_special_event`" when referred to in a handler definition. The name of the method that is annotated with `@handler` is of no significance. You can choose it to your liking. Throughout the circuits source code, handler methods usually follow the pattern "`_on_some_event`". This makes it obvious to the reader that the method is not part of the class's public API (leading underscore as per Python convention) and that it is invoked for events of type `SomeEvent`.

The optional keyword argument "`channel`" can be used to attach the handler to a different channel than the component's channel (as specified by the component's channel attribute).

Handler methods must be declared with arguments and keyword arguments that match the arguments passed to the event upon its creation. Looking at the API for `Started` you'll find that the component that has been started is passed as an argument to its constructor. Therefore, our handler method must declare one argument (line 15).

The handler annotation accepts some more keyword arguments that influence the behavior of the handler and its invocation. Details can be found in the API description of `handler()`.

Automatic handler definition

To easy the implementation of components with (mostly) standard handlers, components can be derived from `Component`. For such classes a `@handler("method_name")` annotation is applied automatically to all method, unless the method's name starts with an underscore or the method has already an explicit `@handler` annotation.

1.3.4 Values

...

1.3.5 Debugging

...

1.3.6 Tools

...

1.3.7 Ticks

...

1.3.8 Futures

...

1.3.9 Manager

...

1.4 The circuits.web Framework

1.4.1 Introduction

circuits.web is a set of components for building high performance HTTP/1.1 and WSGI/1.0 compliant web applications. These components make it easy to rapidly develop rich, scalable web applications with minimal effort.

circuits.web borrows from

- CherryPy

- BaseHTTPServer (*Python std. lib*)
- wsgiref (*Python std. lib*)

Overview

The `circuits.web` namespace contains the following exported components and events for convenience:

Events

Event	Description
Request	The Request Event
Response	The Response Event
Stream	The Stream Event

Servers

Server	Description
BaseServer	The Base Server (no Dispatcher)
Server	The **full** Server + Dispatcher

Error Events

Error	Description
HTTPError	A generic HTTP Error Event
Forbidden	A Forbidden (403) Event
NotFound	A Not Found (404) Event
Redirect	A Redirect (30x) Event

Dispatchers

Dispatcher	Description
Static	A Static File Dispatcher
Dispatcher	The Default Dispatcher
VirtualHosts	Virtual Hosts Dispatcher
XMLRPC	XML-RPC Dispatcher
JSONRPC	JSON-RPC Dispatcher

Other Components

Component	Description
Logger	Default Logger
Controller	Request Handler Mapper
Sessions	Default Sessions Handler

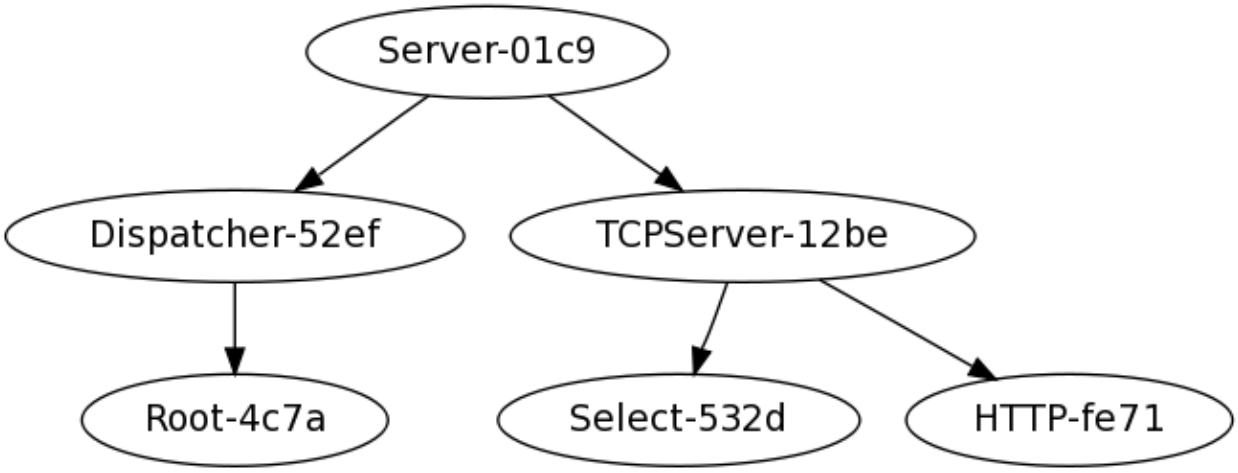
To start working with circuits.web one normally only needs to import from circuits.web, for example:

```
from circuits import Component
from circuits.web import BaseServer

class Root(Component):

    def request(self, request, response):
        return "Hello World!"

(BaseServer(8000) + Root()).run()
```

For further information regarding any of circuits.web's components, events or other modules and functions refer to the API Documentation.

1.4.2 Getting Started

Just like any application or system built with circuits, a circuits.web application follows the standard Component based design and structure whereby functionality is encapsulated in components. circuits.web itself is designed and built in this fashion. For example a circuits.web Server's structure looks like this:



To illustrate the basic steps, we will demonstrate developing your classical "Hello World!" applications in a web-based way with circuits.web

To get started, we first import the necessary components:

```
from circutis.web import Server, Controller
```

Next we define our first Controller with a single Request Handler defined as our index. We simply return "Hello World!" as the response for our Request Handler.

```
class Root(Controller):

   def index(self):
      return "Hello World!"
```

This completes our simple web application which will respond with "Hello World!" when anyone accesses it.

Admittedly this is a stupidly simple web application! But circuits.web is very powerful and plays nice with other tools.

Now we need to run the application:

```
(Server(8000) + Root()).run()
```

That's it! Navigate to: http://127.0.0.1:8000/ and see the result.

Here's the complete code:

```
from circuits.web import Server, Controller

class Root(Controller):

    def index(self):
        return "Hello World!"

(Server(8000) + Root()).run()
```

Have fun!

1.4.3 The Basics

circuits.web is not a **full stack** web framework, rather it is more closely aligned with CherryPy and offers enough functionality to make quickly developing web applications easy and as flexible as possible. circuits.web does not provide features such as:

- builtin Templating
- builtin Database or ORM tools
- **etc**

The functionality that circutis.web *does* provide ensures that circuits.web is fully HTTP/1.1 and WSGI/1.0 compliant and offers all the essential tools you need to build your web application or website.

1.4.4 How To Guide(s)

1.4.5 Application Deployment

1.4.6 Controllers

1.4.7 Dispatchers

1.4.8 Tools

1.5 HowTos

1.5.1 How To: Build a Simple Server

Overview

In this guide we're going to walk through the steps required to build a simple chat server. Users will connect using a standard telnet client and start chatting with other users that are connected.

Prerequisites

- Python

- circuits

Components Used

- Component
- TCPServer

Events Used

- Write

Step 1 - Setting up

Let's start off by importing the components and events we'll need.

```
#!/usr/bin/env python

from circuits import Component
from circuits.net.sockets import TCPServer, Write
```

Step 2 - Building the Server

Next let's define our Server Component with a simple event handler that broadcasts all incoming messages to every connected client. We'll keep a list of clients connected to our server in self._clients.

We need to define three event handlers.

1. An event handler to update our list of connected clients when a new client connects.
2. An event handler to update our list of connected clients when a client has disconnected.
3. An event handler to handle messages from connected clients and broadcast them to every other connected client.

```
class Server(Component):

    def __init__(self, host, port=8000):
        super(Server, self).__init__()

        self._clients = []

        TCPServer((host, port)).register(self)

    def connect(self, sock, host, port):
        self._clients.append(sock)

    def disconnect(self, sock):
        self._clients.remove(sock)

    def read(self, sock, data):
        for client in self._clients:
            if not client == sock:
                self.fire(Write(client, data.strip()))
```

Let's walk through this in details:

1. Create a new Component called Server

2. Define its initialization arguments as `(host, port=8000)`
3. Call the super constructor of the underlying Component (*This is important as all components need to be initialized properly*)
4. Register a `TCPServer` Component and configure it.
5. Create Event Handlers for:
    - Dealing with new connecting clients.
    - Dealing with clients whom have disconnected.
    - Dealing with messages from connected clients.

Step 3 - Running the Server

The last step is simply to create an instance of the `Server` Component and run it (*making sure to configure it with a host and port*).

```
Server("localhost").run()
```

That's it!

Using a standard telnet client try connecting to localhost on port 8000. Try connecting a second client and watch what happens in the 2nd client when you type text into the 1st.

Enjoy!

Source Code

```
#!/usr/bin/env python

from circuits import Component
from circuits.net.sockets import TCPServer, Write

class Server(Component):

    def __init__(self, host, port=8000):
        super(Server, self).__init__()

        self._clients = []

        TCPServer((host, port)).register(self)

    def connect(self, sock, host, port):
        self._clients.append(sock)

    def disconnect(self, sock):
        self._clients.remove(sock)

    def read(self, sock, data):
        for client in self._clients:
            if not client == sock:
                self.fire(Write(client, data.strip()))

Server("localhost").run()
```

Download simple_server.py

1.6 API Reference

1.6.1 circuits.core – Core Functionality

Core

This package contains the essential core parts of the circuits framework.

circuits.core.components – Components

This module defines the BaseComponent and its subclass Component.

Classes

class circuits.core.components.**BaseComponent**(**args*, ***kwargs*)
Bases: circuits.core.manager.Manager

This is the base class for all components in a circuits based application. Components can (and should, except for root components) be registered with a parent component.

BaseComponents can declare methods as event handlers using the handler decoration (see circuits.core.handlers.handler()). The handlers are invoked for matching events from the component's channel (specified as the component's channel attribute).

BaseComponents inherit from circuits.core.manager.Manager. This provides components with the circuits.core.manager.Manager.fireEvent() method that can be used to fire events as the result of some computation.

Apart from the fireEvent() method, the Manager nature is important for root components that are started or run.

Variables **channel** – a component can be associated with a specific channel by setting this attribute. This should either be done by specifying a class attribute *channel* in the derived class or by passing a keyword parameter *channel="..."* to *__init__*. If specified, the component's handlers receive events on the specified channel only, and events fired by the component will be sent on the specified channel (this behavior may be overridden, see Event, fireEvent() and handler()). By default, the channel attribute is set to "*", meaning that events are fired on all channels and received from all channels.

initializes x; see x.__class__.__doc__ for signature

register(*parent*)
Inserts this component in the component tree as a child of the given *parent* node.

Parameters **parent** (Manager) – the parent component after registration has completed.

This method fires a Registered event to inform other components in the tree about the new member.

unregister()
Removes this component from the component tree.

Removing a component from the component tree is a two stage process. First, the component is marked as to be removed, which prevents it from receiving further events, and a PrepareUnregister event is fired. This allows other components to e.g. release references to the component to be removed before it is actually removed from the component tree.

After the processing of the PrepareUnregister event has completed, the component is removed from the tree and an Unregistered event is fired.

class circuits.core.components.**Component**(**args*, ***kwargs*)
Bases: circuits.core.components.BaseComponent

If you use Component instead of BaseComponent as base class for your own component class, then all methods that are not marked as private (i.e: start with an underscore) are automatically decorated as handlers.

The methods are invoked for all events from the component's channel where the event's name matches the method's name.

Components

none

Events

class circuits.core.components.**PrepareUnregister**(**args*, ***kwargs*)
Bases: circuits.core.events.Event

This event is fired when a component is about to be unregistered from the component tree. Unregistering a component actually detaches the complete subtree that the unregistered component is the root of. Components that need to know if they are removed from the main tree (e.g. because they maintain relationships to other components in the tree) handle this event, check if the component being unregistered is one of their ancestors and act accordingly.

Parameters **component** – the component that will be unregistered

in_subtree(*component*)
Convenience method that checks if the given *component* is in the subtree that is about to be detached.

class circuits.core.components.**SingletonError**
Bases: exceptions.Exception

Raised if a Component with the *singleton* class attribute is True.

Functions

none

circuits.core.events – Events

This module defines the basic Event class and common events.

Classes

class circuits.core.events.**Event**(**args*, ***kwargs*)
Bases: circuits.core.events.BaseEvent

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a name attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

none

Events

class `circuits.core.events.`**`Error`**(*type*, *value*, *traceback*, *handler=None*)
Bases: `circuits.core.events.Event`

Error Event

This Event is sent for any exceptions that occur during the execution of an Event Handler that is not SystemExit or KeyboardInterrupt.

Parameters

- **type** (*type*) – type of exception
- **value** (*exceptions.TypeError*) – exception object
- **traceback** (*traceback*) – traceback of exception
- **kwargs** (*dict*) – (Optional) Additional Information

class `circuits.core.events.`**`Failure`**(**args*, ***kwargs*)
Bases: `circuits.core.events.DerivedEvent`

Failure Event

This Event is sent when an error has occurred with the execution of an Event Handlers.

Parameters **event** (*Event*) – The event that failed

class circuits.core.events.**Registered**(*component*, *manager*)
Bases: circuits.core.events.Event

Registered Event

This Event is sent when a Component has registered with another Component or Manager. This Event is only sent iif the Component or Manager being registered with is not itself.

Parameters

- **component** (*Component*) – The Component being registered
- **manager** (*Component or Manager*) – The Component or Manager being registered with

class circuits.core.events.**Signal**(*signal*, *stack*)
Bases: circuits.core.events.Event

Signal Event

This Event is sent when a Component receives a signal.

Parameters

- **signal** – The signal number received.
- **stack** – The interrupted stack frame.

class circuits.core.events.**Started**(*component*)
Bases: circuits.core.events.Event

Started Event

This Event is sent when a Component has started running.

Parameters **component** (*Component or Manager*) – The component that was started

class circuits.core.events.**Stopped**(*component*)
Bases: circuits.core.events.Event

Stopped Event

This Event is sent when a Component has stopped running.

Parameters **component** (*Component or Manager*) – The component that has stopped

class circuits.core.events.**Success**(**args*, ***kwargs*)
Bases: circuits.core.events.DerivedEvent

Success Event

This Event is sent when all handlers (for a particular event) have been executed successfully, see Manager.

Parameters **event** (*Event*) – The event that has completed.

class circuits.core.events.**Unregister**(*component=None*)
Bases: circuits.core.events.Event

Unregister Event

This Event ask for a Component to unregister from its Component or Manager.

class circuits.core.events.**Unregistered**(*component*, *manager*)
Bases: circuits.core.events.Event

Unregistered Event

This Event is sent when a Component has been unregistered from its Component or Manager.

Functions

none

circuits.core.handlers – Handlers

This module define the @handler decorator/function and the HandlesType type.

Classes

class circuits.core.handlers.**HandlerMetaClass**(*name*, *bases*, *ns*)
Bases: type

Components

none

Events

none

Functions

circuits.core.handlers.**handler**(**names*, ***kwargs*)
Creates an Event Handler

This decorator can be applied to methods of classes derived from circuits.core.components.BaseComponent. It marks the method as a handler for the events passed as arguments to the @handler decorator. The events are specified by their name.

The decorated method's arguments must match the arguments passed to the circuits.core.events.Event on creation. Optionally, the method may have an additional first argument named *event*. If declared, the event object that caused the handler to be invoked is assigned to it.

By default, the handler is invoked by the component's root Manager for events that are propagated on the channel determined by the BaseComponent's *channel* attribute. This may be overridden by specifying a different channel as a keyword parameter of the decorator (channel=...).

Keyword argument priority influences the order in which handlers for a specific event are invoked. The higher the priority, the earlier the handler is executed.

A handler may also be specified as a filter by adding the keyword argument filter=True to the decorator. If such a handler returns a value different from None, no more handlers are invoked for the handled event. Filtering handlers are invoked before normal handlers with the same priority (but after any handlers with higher priority).

If you want to override a handler defined in a base class of your component, you must specify override=True, else your method becomes an additional handler for the event.

Finally, a handler may be defined as a "tick"-handler by specifying tick=True. Such a handler is invoked at regular intervals ("polling").

Return value

Normally, the results returned by the handlers for an event are simply collected in the `circuits.core.events.Event`'s `value` attribute. As a special case, a handler may return a `types.GeneratorType`. This signals to the dispatcher that the handler isn't ready to deliver a result yet. Rather, it has interrupted it's execution with a `yield None` statement, thus preserving its current execution state.

The dispatcher saves the returned generator object as a task. All tasks are reexamined (i.e. their `next()` method is invoked) when the pending events have been executed.

This feature avoids an unnecessarily complicated chaining of event handlers. Imagine a handler A that needs the results from firing an event E in order to complete. Then without this feature, the final action of A would be to fire event E, and another handler for an event `SuccessE` would be required to complete handler A's operation, now having the result from invoking E available (actually it's even a bit more complicated).

Using this "suspend" feature, the handler simply fires event E and then yields `None` until e.g. it finds a result in E's `value` attribute. For the simplest scenario, there even is a utility method `circuits.core.manager.Manager.callEvent()` that combines firing and waiting.

`circuits.core.manager` – Manager

This module defines the Manager class.

Classes

class `circuits.core.manager.`**`Manager`**(**args*, ***kwargs*)

Bases: `object`

The manager class has two roles. As a base class for component implementation, it provides methods for event and handler management. The method `fireEvent()` appends a new event at the end of the event queue for later execution. `waitEvent()` suspends the execution of a handler until all handlers for a given event have been invoked. `callEvent()` combines the last two methods in a single method.

The methods `addHandler()` and `removeHandler()` allow handlers for events to be added and removed dynamically. (The more common way to register a handler is to use the `handler()` decorator or derive the class from `Component`.)

In its second role, the `Manager` takes the role of the event executor. Every component hierarchy has a root component that maintains a queue of events. Firing an event effectively means appending it to the event queue maintained by the root manager. The `flush()` method removes all pending events from the queue and, for each event, invokes all the handlers. Usually, `flush()` is indirectly invoked by `run()`.

The manager optionally provides information about the execution of events as automatically generated events. If an `Event` has its `success` attribute set to True, the manager fires a `Success` event if all handlers have been executed without error. Note that this event will be enqueued (and dispatched) immediately after the events that have been fired by the event's handlers. So the success event indicates both the successful invocation of all handlers for the event and the processing of the immediate follow-up events fired by those handlers.

Sometimes it is not sufficient to know that an event and its immediate follow-up events have been processed. Rather, it is important to know when all state changes triggered by an event, directly or indirectly, have been performed. This also includes the processing of events that have been fired when invoking the handlers for the follow-up events and the processing of events that have again been fired by those handlers and so on. The completion of the processing of an event and all its direct or indirect follow-up events may be indicated by a `Complete` event. This event is generated by the manager if `Event` has its `complete` attribute set to True.

Apart from the event queue, the root manager also maintains the list of "tick"-handlers, which are to be invoked at regular intervals (see `handler()`) and a list of tasks, actually Python generators, that are updated when the event queue has been flushed.

initializes x; see x.__class__.__doc__ for signature

name
Return the name of this Component/Manager

running
Return the running state of this Component/Manager

fireEvent(*event*, **channels*)
Fire an event into the system.

Parameters

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

fire(*event*, **channels*)
Fire an event into the system.

Parameters

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

callEvent(*event*, **channels*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This method may only be invoked as argument to a `yield` on the top execution level of a handler (e.g. "`yield self.callEvent(event)`"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see `circuits.core.handlers.handler()`).

call(*event*, **channels*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This method may only be invoked as argument to a `yield` on the top execution level of a handler (e.g. "`yield self.callEvent(event)`"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see `circuits.core.handlers.handler()`).

flushEvents()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

flush()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

start(*process=False*)
Start a new thread or process that invokes this manager's `run()` method. The invocation of this method returns immediately after the task or process has been started.

stop()
Stop this manager. Invoking this method either causes an invocation of `run()` to return or terminates the thread or process associated with the manager.

tick(*timeout=-1*)
Execute all possible actions once. Check for any registered tick handlers and run them, process all regis-

tered tasks and flush the event queue. If the application is running fire a GenerateEvents to get new events from sources.

This method is usually invoked from `run()`. It may also be used to build an application specific main loop.

Parameters **timeout** (*float, measuring seconds*) – the maximum waiting time spent in this method. If negative, the method may block until at least one action has been taken.

run()

Run this manager. The method fires the `Started` event and then continuously calls `tick()`.

The method returns when the manager's `stop()` method is invoked.

If invoked by a programs main thread, a signal handler for the `INT` and `TERM` signals is installed. This handler fires the corresponding `Signal` events and then calls `stop()` for the manager.

Components

none

Events

none

Functions

none

circuits.core.values – Value

This defines the Value object used by components and events.

Classes

class circuits.core.values.**Value**(*event=None*, *manager=None*, *notify=False*)

Bases: object

Create a new future Value Object

Creates a new future Value Object which is used by Event Objects and the Manager to store the result(s) of an Event Handler's exeuction of some Event in the system.

Parameters

- **event** (*Event instance*) – The Event this Value is associated with.
- **manager** (*A Manager/Component instance.*) – The Manager/Component used to trigger notifications.
- **notify** (*bool*) – True to notify of changes to this value

Variables

- **result** – True if this value has been changed.
- **errors** – True if while setting this value an exception occured.

This is a Future/Promise implementation.

Components

none

Events

none

Functions

none

circuits.core.futures – Futures

Classes

none

Components

none

Events

none

Functions

circuits.core.futures.**future**(*pool=None*)

Decorator to wrap an event handler in a future Task

This decorator wraps an event handler into a background Task executing the event handler function in the background.

Parameters **pool** (*Pool*) – An optional thread/process pool

circuits.core.pollers – I/O Pollers

Classes

none

Components

class circuits.core.pollers.**BasePoller**(*timeout=0.01*, *channel=None*)
Bases: circuits.core.components.BaseComponent

class circuits.core.pollers.**Select**(*timeout=0.01*, *channel='select'*)
Bases: circuits.core.pollers.BasePoller

Select(...) -> new Select Poller Component

Creates a new Select Poller Component that uses the select poller implementation. This poller is not recommended but is available for legacy reasons as most systems implement select-based polling for backwards compatibility.

class circuits.core.pollers.**Poll**(*timeout=0.01*, *channel='poll'*)
Bases: circuits.core.pollers.BasePoller

Poll(...) -> new Poll Poller Component

Creates a new Poll Poller Component that uses the poll poller implementation.

class circuits.core.pollers.**EPoll**(*timeout=0.01*, *channel='epoll'*)
Bases: circuits.core.pollers.BasePoller

EPoll(...) -> new EPoll Poller Component

Creates a new EPoll Poller Component that uses the epoll poller implementation.

Events

none

Functions

none

circuits.core.pools – Worker Pools

Pools

Thread and Process based "worker" pools.

Events

none

Components

class circuits.core.pools.**Pool**(*min=5*, *max=10*, *processes=False*, *channel='pool'*)
Bases: circuits.core.components.BaseComponent

Functions

none

circuits.core.timers – Timers

Timer component to facilitate timed events.

Classes

none

Components

class circuits.core.timers.**Timer**(*interval*, *event*, **channels*, ***kwargs*)
Bases: circuits.core.components.BaseComponent

A timer is a component that fires an event once after a certain delay or periodically at a regular interval.

Parameters

- **interval** (*datetime or float number*) – the delay or interval to wait for until the event is fired. If interval is specified as datetime, the interval is recalculated as the time span from now to the given datetime.
- **event** (Event) – the event to fire.

If the optional keyword argument *persist* is set to True, the event will be fired repeatedly. Else the timer fires the event once and then unregisters itself.

reset()
Reset the timer, i.e. clear the amount of time already waited for.

Events

none

Functions

none

circuits.core.loader – Loader

This module implements a generic Loader suitable for dynamically loading components from other modules. This supports loading from local paths, eggs and zip archives. Both setuptools and distribute are fully supported.

Classes

none

Components

class circuits.core.loader.**Loader**(*auto_register=True*, *init_args=None*, *init_kwargs=None*, *paths=None*, *channel='loader'*)

Bases: circuits.core.components.BaseComponent

Create a new Loader Component

Creates a new Loader Component that enables dynamic loading of components from modules either in local paths, eggs or zip archives.

initializes x; see x.__class__.__doc__ for signature

Events

none

Functions

none

circuits.core.utils – Utilities

Utils

This module defines utilities used by circuits.

Classes

none

Components

none

Events

none

Functions

circuits.core.utils.**findchannel**(*root*, *channel*, *all=False*)

circuits.core.utils.**findcmp**(*root*, *component*, *all=False*)

circuits.core.utils.**findroot**(*component*)

circuits.core.utils.**findtype**(*root*, *component*, *all=False*)

circuits.core.utils.**flatten**(*root*, *visited=None*)

circuits.core.utils.**safeimport**(*name*)

circuits.core.workers – Workers

Workers

Worker components used to perform "work" in independent threads or processes. Worker(s) are typically used by a Pool (circuits.core.pools) to create a pool of workers. Worker(s) are not registered with a Manager or another Component - instead they are managed by the Pool. If a Worker is used independently it should not be registered as it causes its main event handler `_on_task` to execute in the other thread blocking it.

Classes

none

Components

class circuits.core.workers.**Worker**(*process=False*, *channel='worker'*)
Bases: circuits.core.components.BaseComponent

A thread/process Worker Component

This Component creates a Worker (either a thread or process) which when given a `Task`, will execute the given function in the task in the background in its thread/process.

Parameters **process** (*bool*) – True to start this Worker as a process (Thread otherwise)

Events

class circuits.core.workers.**Task**(*f*, **args*, ***kwargs*)
Bases: circuits.core.events.Event

Task Event

This Event is used to initiate a new task to be performed by a Worker or a Pool of Worker(s).

Parameters

- **f** (*function*) – The function to be executed.
- **args** (*tuple*) – Arguments to pass to the function
- **kwargs** (*dict*) – Keyword Arguments to pass to the function

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Functions

none

circuits.core.debugger – Debugger

Debugger component used to debug each event in a system by printing each event to sys.stderr or to a Logger Component instance.

Classes

none

Components

class circuits.core.debugger.**Debugger**(*errors=True*, *events=True*, *file=None*, *logger=None*, *prefix=None*, *trim=None*, ***kwargs*)

Bases: circuits.core.components.BaseComponent

Create a new Debugger Component

Creates a new Debugger Component that filters all events in the system printing each event to sys.stderr or a Logger Component.

Variables

- **IgnoreEvents** – list of events (str) to ignore
- **IgnoreChannels** – list of channels (str) to ignore
- **enabled** – Enabled/Disabled flag

Parameters **log** – Logger Component instance or None (*default*)

initializes x; see x.__class__.__doc__ for signature

Events

none

Functions

none

1.6.2 circuits.io – I/O Support

I/O Support

This package contains various I/O Components. Provided are a a generic File Component, StdIn, StdOut and StdErr components. Instances of StdIn, StdOUt and StdErr are also created by importing this package.

circuits.io.events – I/O Events

Events

class circuits.io.events.**Close**(**args*, ***kwargs*)

Bases: circuits.core.events.Event

Close Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class `circuits.io.events.`**`Closed`**(**args*, ***kwargs*)

Bases: `circuits.core.events.Event`

Closed Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.

- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class `circuits.io.events.`**`EOF`**(**args*, ***kwargs*)

Bases: `circuits.core.events.Event`

EOF Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class `circuits.io.events.`**`Error`**(**args*, ***kwargs*)

Bases: `circuits.core.events.Event`

Error Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class `circuits.io.events.`**`Opened`**(**args*, ***kwargs*)
Bases: `circuits.core.events.Event`

Opened Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class `circuits.io.events.`**Read**(**args*, ***kwargs*)

Bases: `circuits.core.events.Event`

Read Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class circuits.io.events.**Seek**(**args*, ***kwargs*)
Bases: circuits.core.events.Event

Seek Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a circuits.core.values.Value object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class circuits.io.events.**Write**(**args*, ***kwargs*)
Bases: circuits.core.events.Event

Write Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

none

Functions

none

`circuits.io.file` – File I/O

Events

none

Components

class `circuits.io.file.`**`File`**(*filename=None*, *mode='r'*, *fd=None*, *autoclose=True*, *bufsize=4096*, *encoding='utf-8'*, *channel='file'*)

Bases: `circuits.core.components.Component`

Functions

none

circuits.io.notify – File System Notification

Events

Components

Functions

circuits.io.serial – Serial I/O

Events

none

Components

class circuits.io.serial.**Serial**(*port*, *baudrate=115200*, *bufsize=4096*, *timeout=0.2*, *channel='serial'*)

Bases: circuits.core.components.Component

Functions

none

1.6.3 circuits.net – Networking

Networking Components

This package contains components that implement network sockets and protocols for implementing client and server network applications.

copyright CopyRight (C) 2004-2012 by James Mills

license MIT (See: LICENSE)

circuits.net.protocols – Networking Protocols

Networking Protocols

This package contains components that implement various networking protocols.

circuits.net.protocols.http – HTTP Protocol

Events

class circuits.net.protocols.http.**Request**(**args*, ***kwargs*)

Bases: circuits.core.events.Event

Request Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class `circuits.net.protocols.http.`**`Response`**(**args*, ***kwargs*)

Bases: `circuits.core.events.Event`

Response Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.

- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

class `circuits.net.protocols.http.`**`HTTP`**(*encoding='utf-8'*, *channel='web'*)

Bases: `circuits.core.components.BaseComponent`

Functions

`circuits.net.protocols.irc` – IRC Protocol

Events

class `circuits.net.protocols.irc.`**`RAW`**(**args*, ***kwargs*)

Bases: `circuits.net.protocols.irc.Command`

RAW command

Components

class `circuits.net.protocols.irc.`**`IRC`**(**args*, ***kwargs*)

Bases: `circuits.core.components.Component`

IRC Protocol Component

Creates a new IRC Component instance that implements the IRC Protocol. Incoming messages are handled by the "read" Event Handler, parsed and processed with appropriate Events created and exposed to the rest of te system to listen to and handle.

@note: This Component must be used in conjunction with a Component that exposes Read Events on a "read" Channel.

`line`(*line*)

Line Event Handler

Process a line of text and generate the appropiate event. This must not be overridden by sub-classes, if it is, this must be explitetly called by the sub-class. Other Components may however listen to this event and process custom IRC events.

Functions

circuits.net.protocols.line – Line Protocol

Events

class circuits.net.protocols.line.**Line**(**args*, ***kwargs*)

Bases: circuits.core.events.Event

Line Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a circuits.core.values.Value object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

class circuits.net.protocols.line.**LP**(**args*, ***kwargs*)

Bases: circuits.core.components.BaseComponent

Line Protocol

Implements the Line Protocol.

Incoming data is split into lines with a splitter function. For each line of data processed a Line Event is created. Any unfinished lines are appended into an internal buffer.

A custom line splitter function can be passed to customize how data is split into lines. This function must accept two arguments, the data to process and any left over data from a previous invocation of the splitter function. The function must also return a tiple of two items, a list of lines and any left over data.

Parameters **splitter** (*function*) – a line splitter function

This Component operates in two modes. In normal operation it's expected to be used in conjunction with components that expose a Read Event on a "read" channel with only one argument (data). Some builtin components that expose such events are: - circuits.net.sockets.TCPClient - circuits.io.File

The second mode of operation works with circuits.net.sockets.Server components such as TCPServer, UNIXServer, etc. It's expected that two arguments exist in the Read Event, sock and data. The following two arguments can be passed to affect how unfinished data is stored and retrieved for such components:

Parameters **getBuffer** (*function*) – function to retrieve the buffer for a client sock

This function must accept one argument (sock,) the client socket whoose buffer is to be retrieved.

Parameters **updateBuffer** (*function*) – function to update the buffer for a client sock

This function must accept two arguments (sock, buffer,) the client socket and the left over buffer to be updated.

@note: This Component must be used in conjunction with a Component that exposes Read events on a "read" Channel.

initializes x; see x.__class__.__doc__ for signature

Functions

`circuits.net.protocols.http` – HTTP Protocol

Events

class `circuits.net.protocols.http.`**`Request`**(**args*, ***kwargs*)
Bases: `circuits.core.events.Event`

Request Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class `circuits.net.protocols.http.`**`Response`**(**args*, ***kwargs*)

Bases: `circuits.core.events.Event`

Response Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

class `circuits.net.protocols.http.`**`HTTP`**(*encoding='utf-8'*, *channel='web'*)

Bases: `circuits.core.components.BaseComponent`

Functions

circuits.net.protocols.irc – IRC Protocol

Events

class circuits.net.protocols.irc.**RAW**(**args*, ***kwargs*)
Bases: circuits.net.protocols.irc.Command

RAW command

Components

class circuits.net.protocols.irc.**IRC**(**args*, ***kwargs*)
Bases: circuits.core.components.Component

IRC Protocol Component

Creates a new IRC Component instance that implements the IRC Protocol. Incoming messages are handled by the "read" Event Handler, parsed and processed with appropriate Events created and exposed to the rest of te system to listen to and handle.

@note: This Component must be used in conjunction with a Component that exposes Read Events on a "read" Channel.

line(*line*)
Line Event Handler

Process a line of text and generate the appropiate event. This must not be overridden by sub-classes, if it is, this must be explitetly called by the sub-class. Other Components may however listen to this event and process custom IRC events.

Functions

circuits.net.protocols.line – Line Protocol

Events

class circuits.net.protocols.line.**Line**(**args*, ***kwargs*)
Bases: circuits.core.events.Event

Line Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a name attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to

as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

class `circuits.net.protocols.line.`**`LP`**(**args*, ***kwargs*)

Bases: `circuits.core.components.BaseComponent`

Line Protocol

Implements the Line Protocol.

Incoming data is split into lines with a splitter function. For each line of data processed a Line Event is created. Any unfinished lines are appended into an internal buffer.

A custom line splitter function can be passed to customize how data is split into lines. This function must accept two arguments, the data to process and any left over data from a previous invocation of the splitter function. The function must also return a tiple of two items, a list of lines and any left over data.

Parameters **splitter** (*function*) – a line splitter function

This Component operates in two modes. In normal operation it's expected to be used in conjunction with components that expose a Read Event on a "read" channel with only one argument (data). Some builtin components that expose such events are: - circuits.net.sockets.TCPClient - circuits.io.File

The second mode of operation works with circuits.net.sockets.Server components such as TCPServer, UNIXServer, etc. It's expected that two arguments exist in the Read Event, sock and data. The following two arguments can be passed to affect how unfinished data is stored and retrieved for such components:

Parameters **getBuffer** (*function*) – function to retrieve the buffer for a client sock

This function must accept one argument (sock,) the client socket whoose buffer is to be retrieved.

Parameters **updateBuffer** (*function*) – function to update the buffer for a client sock

This function must accept two arguments (sock, buffer,) the client socket and the left over buffer to be updated.

@**note: This Component must be used in conjunction with a Component that** exposes Read events on a "read" Channel.

initializes x; see x.__class__.__doc__ for signature

Functions

circuits.net.sockets – Socket Components

Events

class circuits.net.sockets.**Connect**(**args*, ***kwargs*)
Bases: circuits.core.events.Event

Connect Event

This Event is sent when a new client connection has arrived on a server. This event is also used for client's to initiate a new connection to a remote host.

Note: This event is used for both Client and Server Components.

Parameters

- **args** (*tuple*) – Client: (host, port) Server: (sock, host, port)
- **kwargs** (*dict*) – Client: (ssl)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

class circuits.net.sockets.**Connected**(*host*, *port*)
Bases: circuits.core.events.Event

Connected Event

This Event is sent when a client has successfully connected.

@note: This event is for Client Components.

Parameters

- **host** – The hostname connected to.
- **port** – The port connected to

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

class circuits.net.sockets.**Close**(**args*)
Bases: circuits.core.events.Event

Close Event

This Event is used to notify a client, client connection or server that we want to close.

@note: This event is never sent, it is used to close. @note: This event is used for both Client and Server Components.

Parameters **args** – Client: () Server: (sock)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

class circuits.net.sockets.**Closed**
Bases: circuits.core.events.Event

Closed Event

This Event is sent when a server has closed its listening socket.

@note: This event is for Server components.

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

class circuits.net.sockets.**Read**(**args*)
Bases: circuits.core.events.Event

Read Event

This Event is sent when a client or server connection has read any data.

@note: This event is used for both Client and Server Components.

Parameters **args** – Client: (data) Server: (sock, data)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

class circuits.net.sockets.**Write**(**args*)
Bases: circuits.core.events.Event

Write Event

This Event is used to notify a client, client connection or server that we have data to be written.

@note: This event is never sent, it is used to send data. @note: This event is used for both Client and Server Components.

Parameters **args** – Client: (data) Server: (sock, data)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

class circuits.net.sockets.**Error**(**args*)
Bases: circuits.core.events.Event

Error Event

This Event is sent when a client or server connection has an error.

@note: This event is used for both Client and Server Components.

Parameters **args** – Client: (error) Server: (sock, error)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

class circuits.net.sockets.**Disconnect**(**args*)
Bases: circuits.core.events.Event

Disconnect Event

This Event is sent when a client connection has closed on a server. This event is also used for client's to disconnect from a remote host.

@note: This event is used for both Client and Server Components.

Parameters **args** – Client: () Server: (sock)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

class circuits.net.sockets.**Disconnected**
Bases: circuits.core.events.Event

Disconnected Event

This Event is sent when a client has disconnected

@note: This event is for Client Components.

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Components

class circuits.net.sockets.**Client**(*bind=None*, *bufsize=4096*, *channel='client'*)
Bases: circuits.core.components.Component

channel = 'client'

connected

close()

write(*data*)

class circuits.net.sockets.**TCPClient**(*bind=None*, *bufsize=4096*, *channel='client'*)
Bases: circuits.net.sockets.Client

connect(*host*, *port*, *secure=False*, ***kwargs*)

circuits.net.sockets.**UDPClient**
alias of UDPServer

class circuits.net.sockets.**UNIXClient**(*bind=None*, *bufsize=4096*, *channel='client'*)
Bases: circuits.net.sockets.Client

ready(*component*)

connect(*path*, *secure=False*, ***kwargs*)

class circuits.net.sockets.**Server**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: circuits.core.components.Component

channel = 'server'

connected

host

port

close(*sock=None*)

write(*sock*, *data*)

broadcast(*data*)

class circuits.net.sockets.**TCPServer**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: circuits.net.sockets.Server

class circuits.net.sockets.**UDPServer**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: circuits.net.sockets.Server

close()

write(*address*, *data*)

broadcast(*data*, *port*)

class circuits.net.sockets.**UNIXServer**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: circuits.net.sockets.Server

Functions

circuits.net.sockets.**Pipe**(**channels*, ***kwargs*)
Create a new full duplex Pipe

Returns a pair of UNIXClient instances connected on either side of the pipe.

1.6.4 circuits.node – Node

Node

Distributed and Inter-Processing support for circuits

circuits.node.client – Client

Client

...

Events

none

Components

class circuits.node.client.**Client**(*host*, *port*, *channel='node'*)
Bases: circuits.core.components.BaseComponent

Client

...

Functions

none

circuits.node.events – Events

Events

...

Events

class circuits.node.events.**Packet**(**args*, ***kwargs*)
Bases: circuits.core.events.Event

Packet Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class `circuits.node.events.`**`Remote`**(*event*, *node*, *channel=None*)

Bases: `circuits.core.events.Event`

Remote Event

...

Components

none

Functions

none

`circuits.node.node` – Node

Node

...

Events

none

Components

class circuits.node.node.**Node**(*bind=None*, *channel='node'*)
Bases: circuits.core.components.BaseComponent

Node

...

Functions

none

circuits.node.server – Server

Server

...

Events

none

Components

class circuits.node.server.**Server**(*bind*, *channel='node'*)
Bases: circuits.core.components.BaseComponent

Server

...

Functions

none

circuits.node.utils – Node Utilities

Utils

...

Events

none

Components

none

Functions

circuits.node.utils.**dump_event**(*e*, *id*)

circuits.node.utils.**dump_value**(*v*)

circuits.node.utils.**load_event**(*s*)

circuits.node.utils.**load_value**(*v*)

1.6.5 circuits.tools – Development Tools

Often you will end up needing to do some debugging or inspection of your system. The *circuits.tools* package provides a set of development tools for debugging, inspection and analysis.

circuits.tools.**graph**(*x*, *name=None*)

Display a directed graph of the Component structure of x

Parameters

- **x** (*Component or Manager*) – A Component or Manager to graph
- **name** (*str*) – A name for the graph (defaults to x's name)

@return: A directed graph representing x's Component sturcture. @rtype: str

circuits.tools.**reprhandler**(*handler*)

1.6.6 circuits.web – Web Framework

Circuits Library - Web

circuits.web contains the circuits full stack web server that is HTTP and WSGI compliant.

circuits.web.client – Client

Events

class circuits.web.client.**Request**(*method*, *path*, *body=None*, *headers={}*)

Bases: circuits.core.events.Event

Request Event

This Event is used to initiate a new request.

Parameters

- **method** (*str*) – HTTP Method (PUT, GET, POST, DELETE)
- **path** (*str*) – Path to resource

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Components

class circuits.web.client.**Client**(*url*, *channel='client'*)
Bases: circuits.core.components.BaseComponent

channel = 'client'

write(*data*)

close()

connect()

request(*method*, *path*, *body=None*, *headers={}*)

connected

response

Functions

circuits.web.constants – Global Constants

Global Constants

This module implements required shared global constants.

circuits.web.controllers – Controllers

Controllers

This module implements ...

Events

none

Components

class circuits.web.controllers.**Controller**(**args*, ***kwargs*)
Bases: circuits.web.controllers.BaseController

initializes x; see x.__class__.__doc__ for signature

class circuits.web.controllers.**JSONController**(**args*, ***kwargs*)
Bases: circuits.web.controllers.BaseController

initializes x; see x.__class__.__doc__ for signature

Functions

circuits.web.controllers.**expose**(**channels*, ***config*)

circuits.web.controllers.**exposeJSON**(**channels*, ***config*)

circuits.web.dispatchers.dispatcher – Default Dispatcher

Events

none

Components

class circuits.web.dispatchers.dispatcher.**Dispatcher**(***kwargs*)
Bases: circuits.core.components.BaseComponent

Functions

none

circuits.web.dispatchers.jsonrpc – JSON-RPC

JSON RPC

This module implements a JSON RPC dispatcher that translates incoming RPC calls over JSON into RPC events.

Events

class circuits.web.dispatchers.jsonrpc.**RPC**(**args*, ***kwargs*)
Bases: circuits.core.events.Event

RPC Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- **value** – this is a circuits.core.values.Value object that holds the results returned by the handlers invoked for the event.

- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

class `circuits.web.dispatchers.jsonrpc.`**`JSONRPC`**(*path=None*, *encoding='utf-8'*, *rpc_channel='*'*)

Bases: `circuits.core.components.BaseComponent`

Functions

none

`circuits.web.dispatchers.static` – Static

Static

This modStatic implements a Static dispatcher used to serve up static resources and an optional apache-style directory listing.

Events

none

Components

class `circuits.web.dispatchers.static.`**`Static`**(*path=None*, *docroot=None*, *defaults=('index.html', 'index.xhtml')*, *dirlisting=False*)

Bases: `circuits.core.components.BaseComponent`

Functions

none

`circuits.web.dispatchers.virtualhosts` – Virtual Hosts

VirtualHost

This module implements a virtual host dispatcher that sends requests for configured virtual hosts to different dispatchers.

Events

none

Components

class circuits.web.dispatchers.virtualhosts.**VirtualHosts**(*domains*)
Bases: circuits.core.components.BaseComponent

Forward to anotehr Dispatcher based on the Host header.

This can be useful when running multiple sites within one server. It allows several domains to point to different parts of a single website structure. For example: - http://www.domain.example -> / - http://www.domain2.example -> /domain2 - http://www.domain2.example:443 -> /secure

Parameters **domains** (*dict*) – a dict of {host header value: virtual prefix} pairs.

The incoming "Host" request header is looked up in this dict, and, if a match is found, the corresponding "virtual prefix" value will be prepended to the URL path before passing the request onto the next dispatcher.

Note that you often need separate entries for "example.com" and "www.example.com". In addition, "Host" headers may contain the port number.

Functions

none

circuits.web.dispatchers.websockets – WebSockets

WebSockets

This module implements a WebSockets dispatcher that handles the WebSockets handshake, upgrades the connection and translates incoming messages into Message events.

Events

class circuits.web.dispatchers.websockets.**Message**(**args*, ***kwargs*)
Bases: circuits.core.events.Event

Message Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a name attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

class `circuits.web.dispatchers.websockets.`**`WebSockets`**(*path=None*, *wschannel='ws'*)
Bases: `circuits.core.components.BaseComponent`

Functions

none

`circuits.web.dispatchers.xmlrpc` – XML-RPC

XML RPC

This module implements a XML RPC dispatcher that translates incoming RPC calls over XML into RPC events.

Events

class `circuits.web.dispatchers.xmlrpc.`**`RPC`**(**args*, ***kwargs*)
Bases: `circuits.core.events.Event`

RPC Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

class `circuits.web.dispatchers.xmlrpc.`**`XMLRPC`**(*path=None*, *encoding='utf-8'*, *rpc_channel='*'*)

Bases: `circuits.core.components.BaseComponent`

Functions

none

`circuits.web.errors` – Errors

Errors

This module implements a set of standard HTTP Errors.

Events

class `circuits.web.errors.`**`Forbidden`**(*request*, *response*, *code=None*, ***kwargs*)

Bases: `circuits.web.errors.HTTPError`

class `circuits.web.errors.`**`NotFound`**(*request*, *response*, *code=None*, ***kwargs*)

Bases: `circuits.web.errors.HTTPError`

class `circuits.web.errors.`**`Redirect`**(*request*, *response*, *urls*, *code=None*)

Bases: `circuits.web.errors.HTTPError`

class circuits.web.errors.**Unauthorized**(*request*, *response*, *code=None*, ***kwargs*)
Bases: circuits.web.errors.HTTPError

Components

none

Functions

none

circuits.web.events – Events

Events

class circuits.web.events.**WebEvent**(**args*, ***kwargs*)
Bases: circuits.core.events.Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a circuits.core.values.Value object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class circuits.web.events.**Request**(**args*, ***kwargs*)
Bases: circuits.web.events.WebEvent

Request(WebEvent) -> Request WebEvent

args: request, response

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

classmethod **create**(*name*, **args*, ***kwargs*)
All classes derived dynamically from Request are LiteralEvents.

class circuits.web.events.**Response**(**args*, ***kwargs*)
Bases: circuits.web.events.WebEvent

Response(WebEvent) -> Response WebEvent

args: request, response

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class `circuits.web.events.`**`Stream`**(**args*, ***kwargs*)

Bases: `circuits.web.events.WebEvent`

Stream(WebEvent) -> Stream WebEvent

args: request, response

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

none

Functions

none

`circuits.web.exceptions` – Exceptions

Exceptions

This module implements a set of standard HTTP Errors as Python Exceptions.

Note: This code is mostly borrowed from werkzeug and adapted for circuits.web

Exceptions

exception `circuits.web.exceptions.`**`HTTPException`**(*description=None*, *traceback=None*)
Bases: `exceptions.Exception`

Baseclass for all HTTP exceptions. This exception can be called by WSGI applications to render a default error page or you can catch the subclasses of it independently and render nicer error messages.

exception `circuits.web.exceptions.`**`BadGateway`**(*description=None*, *traceback=None*)
Bases: `circuits.web.exceptions.HTTPException`

502 Bad Gateway

If you do proxying in your application you should return this status code if you received an invalid response from the upstream server it accessed in attempting to fulfill the request.

exception `circuits.web.exceptions.`**`BadRequest`**(*description=None*, *traceback=None*)
Bases: `circuits.web.exceptions.HTTPException`

400 Bad Request

Raise if the browser sends something to the application the application or server cannot handle.

exception `circuits.web.exceptions.`**`Forbidden`**(*description=None*, *traceback=None*)
Bases: `circuits.web.exceptions.HTTPException`

403 Forbidden

Raise if the user doesn't have the permission for the requested resource but was authenticated.

exception circuits.web.exceptions.**Gone**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

410 Gone

Raise if a resource existed previously and went away without new location.

exception circuits.web.exceptions.**InternalServerError**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

500 Internal Server Error

Raise if an internal server error occurred. This is a good fallback if an unknown error occurred in the dispatcher.

exception circuits.web.exceptions.**LengthRequired**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

411 Length Required

Raise if the browser submitted data but no `Content-Length` header which is required for the kind of processing the server does.

exception circuits.web.exceptions.**MethodNotAllowed**(*method*, *description=None*)
Bases: circuits.web.exceptions.HTTPException

405 Method Not Allowed

Raise if the server used a method the resource does not handle. For example *POST* if the resource is view only. Especially useful for REST.

The first argument for this exception should be a list of allowed methods. Strictly speaking the response would be invalid if you don't provide valid methods in the header which you can do with that list.

exception circuits.web.exceptions.**NotAcceptable**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

406 Not Acceptable

Raise if the server can't return any content conforming to the *Accept* headers of the client.

exception circuits.web.exceptions.**NotFound**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

404 Not Found

Raise if a resource does not exist and never existed.

exception circuits.web.exceptions.**NotImplemented**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

501 Not Implemented

Raise if the application does not support the action requested by the browser.

exception circuits.web.exceptions.**PreconditionFailed**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

412 Precondition Failed

Status code used in combination with `If-Match`, `If-None-Match`, or `If-Unmodified-Since`.

exception circuits.web.exceptions.**Redirect**(*urls*, *status=None*)
Bases: circuits.web.exceptions.HTTPException

exception circuits.web.exceptions.**RequestEntityTooLarge**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

413 Request Entity Too Large

The status code one should return if the data submitted exceeded a given limit.

exception circuits.web.exceptions.**RequestTimeout**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

408 Request Timeout

Raise to signalize a timeout.

exception circuits.web.exceptions.**RequestURITooLarge**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

414 Request URI Too Large

Like *413* but for too long URLs.

exception circuits.web.exceptions.**ServiceUnavailable**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

503 Service Unavailable

Status code you should return if a service is temporarily unavailable.

exception circuits.web.exceptions.**Unauthorized**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

401 Unauthorized

Raise if the user is not authorized. Also used if you want to use HTTP basic auth.

exception circuits.web.exceptions.**UnicodeError**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

raised by the request functions if they were unable to decode the incoming data properly.

exception circuits.web.exceptions.**UnsupportedMediaType**(*description=None*, *traceback=None*)
Bases: circuits.web.exceptions.HTTPException

415 Unsupported Media Type

The status code returned if the server is unable to handle the media type the client transmitted.

Events

none

Components

none

Functions

none

circuits.web.headers – Headers

Headers Support

This module implements support for parsing and handling headers.

Events

none

Classes

class circuits.web.headers.**Headers**(*headers*=[ ])
Bases: dict

Manage a collection of HTTP response headers

has_key(*name*)
Return true if the message contains the header.

get_all(*name*)
Return a list of all the values for the named field.

These will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

get(*name*, *default=None*)
Get the first header value for 'name', or return 'default'

keys()
Return a list of all the header field names.

These will be sorted in the order they appeared in the original header list, or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list.

values()
Return a list of all header values.

These will be sorted in the order they appeared in the original header list, or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list.

items()
Get all the header fields and values.

These will be sorted in the order they were in the original header list, or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list.

setdefault(*name*, *value*)
Return first matching header value for 'name', or 'value'

If there is no header named 'name', add a new header with name 'name' and value 'value'.

add_header(*_name*, *_value*, ***_params*)
Extended header setting.

_name is the header field to add. keyword arguments can be used to set additional parameters for the header field, with underscores converted to dashes. Normally the parameter will be added as key="value" unless value is None, in which case only the key will be added.

Example:

h.add_header('content-disposition', 'attachment', filename='bud.gif')

Note that unlike the corresponding 'email.Message' method, this does *not* handle '(charset, language, value)' tuples: all values must be strings or None.

elements (*key*)
Return a list of HeaderElements for the given header (or None).

class circuits.web.headers.**HeaderElement** (*value*, *params=None*)
Bases: object

An element (with parameters) from an HTTP header's element list.

static parse (*elementstr*)
Transform 'token;key=val' to ('token', {'key': 'val'}).

classmethod from_str (*elementstr*)
Construct an instance from a string of the form 'token;key=val'.

class circuits.web.headers.**AcceptElement** (*value*, *params=None*)
Bases: circuits.web.headers.HeaderElement

An element (with parameters) from an Accept* header's element list.

AcceptElement objects are comparable; the more-preferred object will be "less than" the less-preferred object. They are also therefore sortable; if you sort a list of AcceptElement objects, they will be listed in priority order; the most preferred value will be first. Yes, it should have been the other way around, but it's too late to fix now.

qvalue
The qvalue, or priority, of this value.

Components

none

Functions

circuits.web.headers.**header_elements** (*fieldname*, *fieldvalue*)
Return a HeaderElement list from a comma-separated header str.

circuits.web.headers.**parse_headers** (*data*)

circuits.web.http – HTTP Protocol

Hyper Text Transfer Protocol

This module implements the Hyper Text Transfer Protocol or commonly known as HTTP.

Events

none

Components

class circuits.web.http.**HTTP**(*encoding='utf-8'*, *channel='web'*)
Bases: circuits.core.components.BaseComponent

HTTP Protocol Component

Implements the HTTP server protocol and parses and processes incoming HTTP messages creating and sending an appropriate response.

Functions

none

circuits.web.loggers – Loggers

Logger Component

This module implements Logger Components.

Events

none

Components

class circuits.web.loggers.**Logger**(*file=None*, *logger=None*, ***kwargs*)
Bases: circuits.core.components.BaseComponent

Functions

circuits.web.loggers.**formattime**()

circuits.web.main – circuits.web

Main

circutis.web Web Server and Testing Tool.

Shipped application with static and directory index support suitable for quickly serving up simple web pages.

circuits.web.servers – Servers

Web Servers

This module implements the several Web Server components.

Events

none

Components

class circuits.web.servers.**BaseServer**(*bind*, *encoding='utf-8'*, *channel='web'*)
Bases: circuits.core.components.BaseComponent

Create a Base Web Server

Create a Base Web Server (HTTP) bound to the IP Address / Port or UNIX Socket specified by the 'bind' parameter.

Variables **server** – Reference to underlying Server Component

Parameters **bind** (*Instance of int, list, tuple or str*) – IP Address / Port or UNIX Socket to bind to.

The 'bind' parameter is quite flexible with what valid values it accepts.

If an int is passed, a TCPServer will be created. The Server will be bound to the Port given by the 'bind' argument and the bound interface will default (normally to "0.0.0.0").

If a list or tuple is passed, a TCPServer will be created. The Server will be bound to the Port given by the 2nd item in the 'bind' argument and the bound interface will be the 1st item.

If a str is passed and it contains the ':' character, this is assumed to be a request to bind to an IP Address / Port. A TCpServer will thus be created and the IP Address and Port will be determined by splitting the string given by the 'bind' argument.

Otherwise if a str is passed and it does not contain the ':' character, a file path is assumed and a UNIXServer is created and bound to the file given by the 'bind' argument.

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

class circuits.web.servers.**Server**(*bind*, ***kwargs*)
Bases: circuits.web.servers.BaseServer

Create a Web Server

Create a Web Server (HTTP) complete with the default Dispatcher to parse requests and posted form data dispatching to appropriate Controller(s).

See: circuits.web.servers.BaseServer

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Functions

none

circuits.web.sessions – Sessions

Session Components

This module implements Session Components that can be used to store and access persistent information.

Events

none

Components

class circuits.web.sessions.**Sessions**(*name='circuits.session'*, **args*, ***kwargs*)
Bases: circuits.core.components.Component

Functions

none

circuits.web.tools – Tools

Tools

This module implements tools used throughout circuits.web. These tools can also be used within Controlelrs and request handlers.

Events

none

Components

none

Functions

circuits.web.tools.**basic_auth**(*request*, *response*, *realm*, *users*, *encrypt=None*)
Perform Basic Authentication

If auth fails, returns an Unauthorized error with a basic authentication header.

Parameters

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.
- **encrypt** (*callable*) – Callable used to encrypt the password returned from the user-agent. if None it defaults to a md5 encryption.

circuits.web.tools.**check_auth**(*request*, *response*, *realm*, *users*, *encrypt=None*)
Check Authentication

If an authorization header contains credentials, return True, else False.

Parameters

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.
- **encrypt** (*callable*) – Callable used to encrypt the password returned from the user-agent. if None it defaults to a md5 encryption.

circuits.web.tools.**digest_auth**(*request*, *response*, *realm*, *users*)
Perform Digest Authentication

If auth fails, raise 401 with a digest authentication header.

Parameters

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.

circuits.web.tools.**expires**(*request*, *response*, *secs=0*, *force=False*)
Tool for influencing cache mechanisms using the 'Expires' header.

'secs' must be either an int or a datetime.timedelta, and indicates the number of seconds between response.time and when the response should expire. The 'Expires' header will be set to (response.time + secs).

If 'secs' is zero, the 'Expires' header is set one year in the past, and the following "cache prevention" headers are also set: - 'Pragma': 'no-cache' - 'Cache-Control': 'no-cache, must-revalidate'

If 'force' is False (the default), the following headers are checked: 'Etag', 'Last-Modified', 'Age', 'Expires'. If any are already present, none of the above response headers are set.

circuits.web.tools.**gzip**(*response, level=4, mime_types=['text/html', 'text/plain']*)
Try to gzip the response body if Content-Type in mime_types.

response.headers['Content-Type'] must be set to one of the values in the mime_types arg before calling this function.

No compression is performed if any of the following hold:

- The client sends no Accept-Encoding request header
- No 'gzip' or 'x-gzip' is present in the Accept-Encoding header
- No 'gzip' or 'x-gzip' with a qvalue > 0 is present
- The 'identity' value is given with a qvalue > 0.

circuits.web.tools.**serve_download**(*request*, *response*, *path*, *name=None*)
Serve 'path' as an application/x-download attachment.

circuits.web.tools.**serve_file**(*request*, *response*, *path*, *type=None*, *disposition=None*, *name=None*)
Set status, headers, and body in order to serve the given file.

The Content-Type header will be set to the type arg, if provided. If not provided, the Content-Type will be guessed by the file extension of the 'path' argument.

If disposition is not None, the Content-Disposition header will be set to "<disposition>; filename=<name>". If name is None, it will be set to the basename of path. If disposition is None, no Content-Disposition header will be written.

circuits.web.tools.**validate_etags**(*request*, *response*, *autotags=False*)
Validate the current ETag against If-Match, If-None-Match headers.

If autotags is True, an ETag response-header value will be provided from an MD5 hash of the response body (unless some other code has already provided an ETag header). If False (the default), the ETag will not be automatic.

WARNING: the autotags feature is not designed for URL's which allow methods other than GET. For example, if a POST to the same URL returns no content, the automatic ETag will be incorrect, breaking a fundamental use for entity tags in a possibly destructive fashion. Likewise, if you raise 304 Not Modified,

the response body will be empty, the ETag hash will be incorrect, and your application will break. See http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.24

circuits.web.tools.**validate_since**(*request*, *response*)
Validate the current Last-Modified against If-Modified-Since headers.

If no code has set the Last-Modified response header, then no validation will be performed.

circuits.web.utils – Utilities

Utilities

This module implements utility functions.

Events

none

Components

none

Functions

circuits.web.utils.**compress**(*body*, *compress_level*)
Compress 'body' at the given compress_level.

circuits.web.utils.**dictform**(*form*)

circuits.web.utils.**get_ranges**(*headervalue*, *content_length*)
Return a list of (start, stop) indices from a Range header, or None.

Each (start, stop) tuple will be composed of two ints, which are suitable for use in a slicing operation. That is, the header "Range: bytes=3-6", if applied against a Python string, is requesting resource[3:7]. This function will return the list [(3, 7)].

If this function returns an empty list, you should return HTTP 416.

circuits.web.utils.**parse_body**(*request*, *response*, *params*)

circuits.web.utils.**parse_qs**(*query_string*) → dict
Build a params dictionary from a query_string. If keep_blank_values is True (the default), keep values that are blank.

circuits.web.utils.**url**(*request*, *path=''*, *qs=''*, *script_name=None*, *base=None*, *relative=None*)
Create an absolute URL for the given path.

If 'path' starts with a slash ('/'), this will return

- (base + script_name + path + qs).

If it does not start with a slash, this returns

- (base + script_name [+ request.path] + path + qs).

If script_name is None, request will be used to find a script_name, if available.

If base is None, request.base will be used (if available).

Finally, note that this function can be used to obtain an absolute URL for the current request path (minus the querystring) by passing no args. If you call url(qs=request.qs), you should get the original browser URL (assuming no internal redirections).

If relative is False the output will be an absolute URL (including the scheme, host, vhost, and script_name). If True, the output will instead be a URL that is relative to the current request path, perhaps including '..' atoms. If relative is the string 'server', the output will instead be a URL that is relative to the server root; i.e., it will start with a slash.

`circuits.web.wrappers` – Request/Response Objects

Events

none

Classes

class `circuits.web.wrappers.`**`Body`**
Bases: `object`

Response Body

class `circuits.web.wrappers.`**`Host`**(*ip*, *port*, *name=None*)
Bases: `object`

An internet address.

name should be the client's host name. If not available (because no DNS lookup is performed), the IP address should be used instead.

class `circuits.web.wrappers.`**`Request`**(*sock*, *method*, *scheme*, *path*, *protocol*, *qs*)
Bases: `object`

Creates a new Request object to hold information about a request.

Parameters

- **sock** (*socket.socket*) – The socket object of the request.
- **method** (*str*) – The requsted method.
- **scheme** (*str*) – The requsted scheme.
- **path** (*str*) – The requsted path.
- **protocol** (*str*) – The requsted protocol.
- **qs** (*str*) – The query string of the request.

initializes x; see x.__class__.__doc__ for signature

`server` = None
@cvar: A reference to the underlying server

class `circuits.web.wrappers.`**`Response`**(*request*, *encoding='utf-8'*, *code=None*, *message=None*)
Bases: `object`

Response(sock, request) -> new Response object

A Response object that holds the response to send back to the client. This ensure that the correct data is sent in the correct order.

initializes x; see x.__class__.__doc__ for signature

Components

none

Functions

circuits.web.wrappers.**file_generator**(*input*, *chunkSize=4096*)

circuits.web.wsgi – WSGI Support

WSGI Components

This module implements WSGI Components.

Events

none

Components

class circuits.web.wsgi.**Application**
Bases: circuits.core.components.BaseComponent

class circuits.web.wsgi.**Gateway**(*app*, *path='/'*)
Bases: circuits.core.components.BaseComponent

Functions

none

1.6.7 circuits.app – Application Support

Application Components

Contains various components useful for application development and tasks common to applications.

copyright CopyRight (C) 2004-2012 by James Mills

license MIT (See: LICENSE)

circuits.app.config – Application Config

Events

class circuits.app.config.**Load**(**args*, ***kwargs*)
Bases: circuits.app.config.ConfigEvent

Load Config Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a name attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a circuits.core.values.Value object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to True, an associated event EventSuccess (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to True, an associated event EventComplete (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class circuits.app.config.**Save**(**args*, ***kwargs*)
Bases: circuits.app.config.ConfigEvent

Save Config Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a name attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

class `circuits.app.config.`**`Config`**(*filename*, *defaults=None*, *channel='config'*)
Bases: `circuits.core.components.BaseComponent`

Functions

`circuits.app.daemon` – Application Daemon

Events

class `circuits.app.daemon.`**`Daemonize`**(**args*, ***kwargs*)
Bases: `circuits.core.events.Event`

Daemonize Event

This event can be fired to notify the *Daemon* Component to begin the "daemonization" process. This event is (*by default*) used automatically by the *Daemon* Component in its "started" Event Handler (*This behavior can be overridden*).

Arguments: *None*

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.

- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class `circuits.app.daemon.`**`WritePID`**(**args*, ***kwargs*)

Bases: `circuits.core.events.Event`

"WritePID Event

This event can be fired to notify the *Daemon* Component that is should retrive the current process's id (pid) and write it out to the configured path in the *Daemon* Component. This event (*by default*) is used automatically by the *Daemon* Component after the `Daemonize`.

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.
- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

class `circuits.app.daemon.`**`Daemon`**(*pidfile*, *path='/'*, *stdin=None*, *stdout=None*, *stderr=None*, *channel='daemon'*)

Bases: `circuits.core.components.BaseComponent`

Daemon Component

Parameters

- **pidfile** (*str or unicode*) – .pid filename
- **stdin** (*str or unicode*) – filename to log stdin
- **stdout** (*str or unicode*) – filename to log stdout
- **stderr** (*str or unicode*) – filename to log stderr

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Functions

`circuits.app.env` – Application Environment

Events

class `circuits.app.env.`**`Create`**(**args*, ***kwargs*)

Bases: `circuits.app.env.EnvironmentEvent`

Create Environment Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.

- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

class `circuits.app.env.`**`Load`**(**args*, ***kwargs*)

Bases: `circuits.app.env.EnvironmentEvent`

Load Environment Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.

- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

class `circuits.app.env.`**`Environment`**(*path*, *envname*, *channel='env'*)

Bases: `circuits.core.components.BaseComponent`

Base Environment Component

Creates a new environment component that by default only holds configuration and logger components.

This component can be extended to provide more complex system and application environments.

Functions

`circuits.app.log` – Application Logging

Events

class `circuits.app.log.`**`Log`**(**args*, ***kwargs*)

Bases: `circuits.core.events.Event`

Log Event

An Event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a `name` attribute that is used for matching the event with the handlers. By default, the name is the uncameled class name of the event.

Variables

- **channels** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- **value** – this is a `circuits.core.values.Value` object that holds the results returned by the handlers invoked for the event.

- **success** – if this optional attribute is set to `True`, an associated event `EventSuccess` (original name with "Success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- **complete** – if this optional attribute is set to `True`, an associated event `EventComplete` (original name with "Complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **success_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

Components

class `circuits.app.log.`**`Logger`**(*filename*, *name*, *type*, *level*, *channel='logger'*)
Bases: `circuits.core.components.BaseComponent`

Functions

1.7 Developer Docs

So, you'd like to contribute to circuits ins some way ? Got a bug report ? Having problems running the examples ? Having problems getting circuits working in your environment / platform ?

Excellent. Here's what you need to know.

1.7.1 Development Introduction

Here's how we do things in circuits...

1.7.2 Contributing to circuits

Here's how you can contribute to circuits

1.7.3 Testing

Here's how we do testing in circuits:

1.7.4 Reporting Bugs

Found a new bug with circuits ? Great! Here's how you file a new bug report so that we can fix the problem in a timely manner:

1.8 ChangeLog

1.8.1 circuits-2.0.1 20121124

- Fixed `tests/web/test_main.py` which was badly written.
- Fixed a regression test testing the Debugger component

1.8.2 circuits-2.0.0 20121122 (cheetah)

- Fixed circuits.web entry point
- Fixed *tools.reprhandler()* for compatibility with Python-3.3
- Added `*channels` support to waitEvent
- Added example of using .call
- Fixed logic around firing the Daemonize event when registering this component during run-time and after start-up
- Fixed use of reprhandler
- Fixed listening channel for exceptions/errors.
- Fixed channels for Log event.
- Fixed config loading. Fire a Ready event when the Environment is completely ready.
- Added .items(...) method to Config component.
- Added BaseEvent, LiteralEvent, DerivedEvent to the core and circuits name-spaces
- Fixed IRC protocol
- Added has_option to Config component
- Avoid error if user un-registers a component twice.
- Fixed base_url for WebConsole
- Fixed bug with sending Response for a Expect: 100-continue (Closes issue #32)
- Added a new circuits.web test that ensures that large posts > 1024 bytes work
- Updated conf so that doc can be built even if circuits isn't installed
- Updated reference of guide to howtos
- Updated man headers so that they weren't all "Components"
- Fixed all web dispatcher tests
- Fixed XMLRPC dispatcher. Must have a higher priority than the "default" dispatcher in order to coexist with it.
- Fixed unit test for failure response from web *component* (component's handler must have higher priority than default dispatcher if default dispatcher exists).
- Added failure test for web *controller*.
- Fixed JSON dispatcher. Must have a higher priority than the "default" dispatcher in order to coexist with it.
- Fixed vpath traversal. vpath created in reverse ("test_args/1/2/3" became "3/2/1/test_args").
- Fixed evaluation of the Complete event: exclude events fired by other threads during event processing from the set of events to be tracked.

- Don't call tick on components that are waiting to be unregistered.
- Using new PrepareUnregister event to reliably remove sockets from poller.
- Fixes for PrepareUnregister and added test case.
- Added event that informs components about going to be removed from the tree.
- Fixed client request generation (MUST include Host header).
- Fixed channel naming in web.Client to allow several clients (i.e. connections to web sites) to coexist in an application.
- Prevented uncameling of event names that represent web requests. Handlers can now use the last path segment unmodified as handled event's name.
- Fixed the new dispatcher with new tests
- Fixed bug in complete event generation.
- Added optional event signaling the completion of an event and everything that has been caused by it.
- Added the possibility to redirect the success events to other channels.
- Updated documentation to reflect the new "handler suspend" feature.
- Replaced web dispatcher with simpler version
- Added support for x = yield self.callEvent(...)
- Made test_main more reliable
- Removed old BaseManager from playing with GreenletManager. Fixed test_manager_repr
- Fixed the exceptions being thrown for test_eval, but the test still fails
- Added a new failing test - evaluation of promised values
- Removed superfluous .value in test_longwait
- Added support for alllowing future handlers to have a "special" event parameter just like ordinary handlers.
- Fixed test_success
- Fixed test_removeHandler
- Added support for firing Done() and Success() after all done executing.
- Fixed callEvent
- Added 2 failing tests for yield
- Implemented promises which we detect for in circuits.web in cases where an event handler yields. Also only fire _success events after an event is well and truly finished (in the case of yielding event handlers)
- Fixed a bug with value not being set
- Fixed Issue #26
- Added capability of waiting for a specific event name on a specific channel.
- Fixed bug guarding against tasks already removed.
- Implemented Component.init() support whereby one can define an alternative init() without needing to remember to call super(...)
- Fixed Python 3 compatibility with Unicode strings
- Added 99bottles as an example of concurrency. See: http://wiki.python.org/moin/Concurrency/99Bottles

- Removed old-style channel targeting
- Fixed and tested UDP forwarding
- Simplified udpclient example
- Implemented new version of port forwarded. TCP tested.
- Fixed Read events for UDPServer by setting .notify to True.
- Restructured the only How To Guide - Building a Simple Server
- Renamed _get_request_handler to just find_handler
- Removed channels attribute from WebEvents (fix for issue #29).
- Added Eclipse configuration files.
- Fixed uses of deprecated syntax in app.config
- Modified the defaults for channels. Set channels to event.channels, otherwise self.channel defaulting to *
- Fixed uses of deprecated syntax in env
- Fixed a bug with the Redirect event/error in circuits.web where it didn't handle Unicode strings
- fixed the web dispatcher
- Fixed test_poller_reuse test by using the now findtype() utility function
- fixed and adds tests for the web dispatcher
- Moved parseBody out into circuits.web.utils. Other code cleanup
- Added a test for a bug with the dispatcher mehere found.
- Removed itercmp() function. Added findtype() findchannel() and used better variable names. findcmp is an alias of findtype.
- Implemented optional singleton support for components
- Removed the circuits.web *routes* dispatcher as there are no tests for this and Routes dispatcher is broken - re implement at a later stage
- Removal of End feedback event
- Fixed web/test_value.py
- Fixed web futures test
- Simplified and fixed a lot of issues the circuits.bench
- Fixed circuits.web's exceptions tests and handling of exceptions.
- Fixed a potential bug with `circuits.web.wsgi.Application`.
- Modified Manager to only assign a handler return value if it is not None.
- Fixed `*_success` and `*_failure` events fire on `*event.channels` so they go to the right place as expected. Fixed Issue #21
- Removed event equality test and related tests. Seems rather useless and inconsistently used
- Fixed test_gzip circuits.web test. We no longer have a Started event feedback so have to use a filter
- Fixed a corner case where one might be trying to compare an event object with a non-event object
- Fixed the event handling for circuits.web WebSockets Component by separating out the WebSockets handling from the HTTP handling (WebSocketsMediator).

- Fixed use of Value notification in circuits.web for requests.
- Fixed a bunch of examples and tests using deprecated features.
- Fixed the notify io driver and removed Debugger() from test_notify.
- Added man pages for circuits.bench, circuits.sniff and circuits.web
- Wrapped UNIX-specific calls in try/except
- Tidied up examples and removed unused imports
- removed use of coverage module in daemon test
- removed use of coverage module in signals test
- updated .push calls to .fire calls
- Fixed some deprecations warnings
- Added support for multiple webserver with different channels + tests for it
- Added support for silently ignoring errors when writing to stderr from debugger
- Added requirements.txt file containing requirements for building docs on readthedocs.org
- Added link to Read the Docs for circuits
- Updated doc message for success event
- Fixed interrupt handler to allow ^C to be used to quit sample keyecho app
- Removed deprecated modules and functions that were deprecated 1.6
- Deleted old style event success/failure notifiers
- Fixed handling of components being added/removed when looking for ticks
- Fixed bug with `net.Server` .host and .port attributes.
- Deprecated `__tick__`. Event handlers can now be specified as **tick** functions.
- Fixed handler priority inheritance to make sure we get the results in the right harder
- Fixed missing import of sys in circuits.io

1.8.3 circuits-1.6 (oceans) - 20110626

- Added Python 3 support
- 80% Code Coverage
- Added optional greenlet support adding two new primitives. `.waitEvent(...)` and `.callEvent(...)`.
- Added an example WebSockets server using circuits.web
- Added support for specifying a `Poll` instance to use when using the `@future` decorator to create "future" event handlers.
- Added `add_section`, `has_section` and `set` methods to `app.config.Config` Component.
- Added support for running test suite with distutils `python setup.py test`.
- Added a `_on_signal` event handler on the `BaseEnvironment` Component so that environments can be reloaded by listening to `SIGHUP` signals.
- Added support for using absolute paths in `app.env.Environment`.
- Added support in circuits.web `HTTP` protocol to limit the no. of header fragments. This prevents OOM exploits.

- Added a ticks limit to waitEvent
- Added deprecation warnings for .push .add and .remove methods
- NEW `Loader` Component in `circuits.core` for simple plugin support.
- NEW `app.env` and `app.config` modules including a new `app.startup` modules integrating a common startup for applications.
- NEW `KQueue` poller
- Fixed issue 17
- Renamed `circuits.web.main` module to `circuits.web.__main__` so that `python -m circuits.web` just works.
- Fixed `Server.host` and `Server.port` properties in `circuits.net.sockets`.
- Fixed issue 10
- Fixed `app.Daemon` Component to correctly open the stderr file.
- Fixed triggering of `Success` events.
- Fixed duplicate broadcast handler in `UDPServer`
- Fixed duplicate `Disconnect` event from being triggered twice on `Client` socket components.
- Removed dynamic timeout code from `Select` poller.
- Fixed a bug in the circuits.web `HTTP` protocol where headers were not being buffered per client.
- Fixes a missing Event `Closed()` not being triggered for `UDPServer`.
- Make underlying `UDPServer` socket reusable by setting `SO_REUSEADDR`
- Fixes Server socket being discarded twice on close + disconnect
- Socket.write now expects bytes (bytes for python3 and str for python2)
- Better handling of encoding in HTTP Component (allow non utf-8 encoding)
- Always encode HTTP headers in utf-8
- Fixes error after getting socket.ERRCONNREFUSED
- Allows TCPClient to bind to a specific port
- Improved docs
- Handles closing of UDPServer socket when no client is connected
- Adds an un-register handler for components
- Allows utils.kill to work from a different thread
- Fixes bug when handling "*" in channels and targets
- Fixes a bug that could occur when un-registering components
- Fixes for CPU usage problems when using circuits with no I/O pollers and using a Timer for timed events

1.9 Users

1.9.1 Applications written in circuits

- *SahrisWiki <http://sahriswiki.org>* a Wiki / CMS / Blogging Engine.
- *kdb <http://bitbucket.org/prologic/kdb/>* a pluggable IRC Bot Framework.

1.9.2 Applications integrating circuits

- *realXtend <http://realxtend.org/>* Next generation virtual worlds viewer; Naali, for Python components. Worked on by Toni Alatalo and Petri Aura at *Playsign <http://www.playsign.net/>* in collaboration with other realXtend developers.
- *PriceWaterHouseCoopers <http://www.pwc.com/>* TAMS Report Generator <http://www.tams.com.au>‘ written by James Mills and Michael Anton.
- *Mangahelpers <http://mangahelpers.com/>* Profiler for checking our webpages generation times, SQL and cache queries times, project page on <https://launchpad.net/website-profiler>

1.9.3 Other Users

- *MIT Media Lab, <http://media.mit.edu/>* Uses circuits for several internal prototypes

1.10 Contributors

circuits was originally designed, written and primarily maintained by James Mills (http://prologic.shortcircuit.net.au/).

The following users and developers have contributed to circuits:

- Alessio Deiana
- Dariusz Suchojad
- Tim Miller
- Holger Krekel
- Justin Giorgi
- Edwin Marshall
- Alex Mayfield
- Toni Alatalo
- Michael Lipp

Anyone not listed here (*apologies as this list is taken directly from Mercurial's churn command and output*). We appreciate any and all contributions to circuits.

1.11 Frequently Asked Questions

1.12 Glossary

VCS Version Control System, what you use for versioning your source code

1.13 Users

1.13.1 Applications written in circuits

- *SahrisWiki <http://sahriswiki.org>* a Wiki / CMS / Blogging Engine.
- *kdb <http://bitbucket.org/prologic/kdb/>* a pluggable IRC Bot Framework.

1.13.2 Applications integrating circuits

- *realXtend <http://realxtend.org/>* Next generation virtual worlds viewer; Naali, for Python components. Worked on by Toni Alatalo and Petri Aura at *Playsign <http://www.playsign.net/>* in collaboration with other realXtend developers.
- *PriceWaterHouseCoopers <http://www.pwc.com/>* TAMS Report Generator <http://www.tams.com.au>‘ written by James Mills and Michael Anton.
- *Mangahelpers <http://mangahelpers.com/>* Profiler for checking our webpages generation times, SQL and cache queries times, project page on <https://launchpad.net/website-profiler>

1.13.3 Other Users

- *MIT Media Lab, <http://media.mit.edu/>* Uses circuits for several internal prototypes

CHAPTER

TWO

INDICES AND TABLES

- *Index*
- *modindex*
- *search*
- *Glossary*

PYTHON MODULE INDEX

C