

---

# **circuits Documentation**

***Release 2.1.0***

**James Mills**

March 03, 2013



# CONTENTS



**Release** 2.1.0

**Date** March 03, 2013



# OVERVIEW

circuits is a **Lightweight Event** driven and **Asynchronous Application Framework** for the [Python Programming Language](#) with a strong **Component** Architecture.

circuits also includes a lightweight, high performance and scalable HTTP/WSGI compliant web server as well as various I/O and Networking components.

To take full advantage of circuits and its architecture, circuits encourages you to design your application in terms of loosely coupled components. Circuits has a very powerful message passing system that enables components to interact with each other via events. Applications written this way tend to be more maintainable, easier to develop and scale to complex systems.

circuits' **Loosely Coupled Component Architecture** allows for a high level of **Reuse** and **Scalability**. Simpler components can be combined together to form Complex Components and provide higher level functionality and abstraction. Much of the circuits component library is designed and built this way.

- **Documentation:** <http://packages.python.org/circuits> or [Read the Docs](#).
- **Project website:** <https://bitbucket.org/prologic/circuits/>
- **PyPI page:** <http://pypi.python.org/pypi/circuits>





# FEATURES

- event driven
- concurrency support
- component architecture
- asynchronous I/O components
- no required external dependencies
- full featured web framework ([circuits.web](#))
- coroutine based synchronization primitives



# REQUIREMENTS

- circuits has no dependencies beyond the [Python Standard Library](#).



# SUPPORTED PLATFORMS

- Linux, FreeBSD, Mac OS X, Windows
- Python 2.6, 2.7, 3.2, 3.3
- pypy 2.0



# INSTALLATION

The simplest and recommended way to install circuits is with pip. You may install the latest stable release from PyPI with pip:

```
> pip install circuits
```

If you do not have pip, you may use easy\_install:

```
> easy_install circuits
```

Alternatively, you may download the source package from the [PyPi Page](#) or the [Downloads page](#) on the [Website](#); extract it and install using:

```
> python setup.py install
```





# LICENSE

circuits is licensed under the [MIT License](#).



# FEEDBACK

We welcome any questions or feedback about bugs and suggestions on how to improve circuits. Let us know what you think about circuits. [@pythoncircuits](#).

Do you have suggestions for improvement? Then please [Create an Issue](#) with details of what you would like to see. I'll take a look at it and work with you to either incorporate the idea or find a better solution.



# COMMUNITY

There is also a small community of circuits enthusiasts that you may find on the [#circuits IRC Channel](#) on the [FreeNode IRC Network](#) and the [Mailing List](#).

## 8.1 Contents

### 8.1.1 Getting Started

#### 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__
2.1.0
```

Try some of the examples in the `examples/` directory shipped with the distribution or check out some [Applications using circuits](#)

Have fun :)

#### Downloading

##### Latest Stable Release

The latest stable releases can be downloaded from the [Downloads](#) page (*specifically the Tags tab*).

##### 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 https://bitbucket.org/prologic/circuits/
```

For further instructions on how to use Mercurial, please refer to the [Mercurial Book](#).

## Installing

### Installing from a Source Package

*If you have downloaded a source archive, this applies to you.*

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

For other installation options see:

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

### Installing from the Development Repository

*If you have cloned the source code repository, this applies to you.*

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

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

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
```

---

**Note:** You do not need to reinstall if you have installed with `setuptools` via the circuits repository and used `setuptools` to install in “develop” mode.

---

## Requirements and Dependencies

- circuits has no **required** dependencies beyond the [Python Standard Library](#).
- Python: `>= 2.6` or `pypy >= 2.0`

**Supported Platforms** Linux, FreeBSD, Mac OS X, Windows

**Supported Python Versions** 2.6, 2.7, 3.2, 3.3

**Supported pypy Versions** 2.0

### Other Optional Dependencies

These dependencies are not strictly required and only add additional features.

- `pydot` – For rendering component graphs of an application.
- `pyinotify` – For asynchronous file system event notifications and the `circuits.io.notify` module.

## 8.1.2 Tutorial

### 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.

### The Component

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

```
1  #!/usr/bin/env python
2
3  from circuits import Component
4
5  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`.

```
1  #!/usr/bin/env python
2
3  from circuits import Component
4
5  class MyComponent(Component):
6      """My Component"""
7
8  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...

### Event Handlers

Let’s now extend our little example to say “Hello World!” when its started.

```
1  #!/usr/bin/env python
2
3  from circuits import Component
4
5  class MyComponent(Component):
6
7      def started(self, *args):
8          print("Hello World!")
9
10 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.

---

## 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

```
1  #!/usr/bin/env python
2
3  from circuits import Component
4
5  class Bob(Component):
6
7      def started(self, *args):
8          print("Hello I'm Bob!")
9
10 class Fred(Component):
11
12     def started(self, *args):
13         print("Hello I'm Fred!")
14
15 (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!

## 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.

```
1  #!/usr/bin/env python
2
3  from circuits import Component
4  from circuits.tools import graph
```



```

5
6 class Pound(Component):
7
8     def __init__(self):
9         super(Pound, self).__init__()
10
11         self.bob = Bob().register(self)
12         self.fred = Fred().register(self)
13
14     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

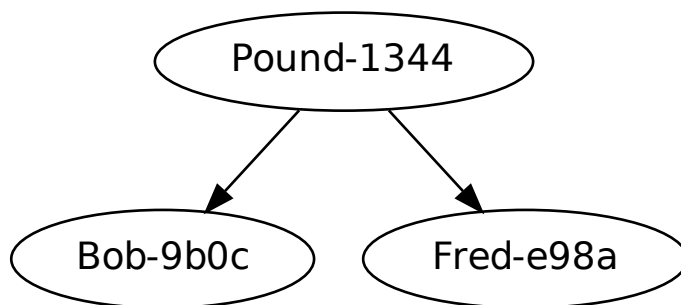
The output of this is identical to the previous:

```

* <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 :-)

## Component Inheritance

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

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.

```
1  #!/usr/bin/env python
2
3  from circuits import Component, Event
4
5
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())
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
32 class Fred(Dog):
33     """Fred"""
34
35 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

## 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:

```
1  #!/usr/bin/env python
2
3  from circuits import Component, Event
4
5
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!
```

## 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.

```
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!" % self.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 008.py

If you run this, you’ll get:

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

## 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 our 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 comes 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()
```

To:

```
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/* 3191:MainThread (queued=0, channels=5, handlers=5) [R]>] {}>
<Registered[fred:registered] [<Fred/fred 3191:MainThread (queued=0, channels=2, handlers=2) [S]>, <Pound/* 3191:MainThread (queued=0, channels=5, handlers=5) [R]>] {}>
<Registered[*:registered] [<Debugger/* 3191:MainThread (queued=0, channels=1, handlers=1) [S]>, <Pound/* 3191:MainThread (queued=0, channels=5, handlers=5) [R]>] {}>
<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 name 'name' is not defined
  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 name 'name' is not defined
  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 an 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\*](#)

## 8.1.3 The circuits Framework

### 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`).

### 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).

```
1  #!/usr/bin/env python
2
3  from circuits import Component, Event
4  from circuits.core.debugger import Debugger
5
6  class Identify(Event):
7      """Identify Event"""
8      success = True
9
10 class Pound(Component):
11
12     def __init__(self):
13         super(Pound, self).__init__()
14
15         Debugger().register(self)
16         Bob().register(self)
17         Fred().register(self)
18
19     def started(self, *args):
20         self.fire(Identify())
21
22     def identify_success(self, evt, result):
23         if not isinstance(result, list):
24             result = [result]
25         print "In pound:"
26         for name in result:
27             print name
```



```

28
29 class Dog(Component):
30
31     def identify(self):
32         return self.__class__.__name__
33
34 class Bob(Dog):
35     """Bob"""
36
37 class Fred(Dog):
38     """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`.

## Handlers

### Explicit Event Handlers

Event Handlers are methods of components that are invoked when a matching event is dispatched. These can be declared explicitly on a `BaseComponent` or `Component` or by using the `handler()` decorator.

```

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 decorator on line 14 turned the method `_on_started` into an event handler for the event `Started`. Event names used to define handlers are the uncamed class names of the event. An event with a class name `MySpecialEvent` becomes “`my_special_event`” when referred to in a handler definition.

When defining explicit event handlers in this way, it's convention to use the following pattern:

```
@handler("foo")
def _on_foo(self, ...):
    ...
```

This makes reading code clear and concise and 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(...)` decorator accepts other keyword arguments that influence the behavior of the event handler and its invocation. Details can be found in the API description of `handler()`.

### Implicit Event Handlers

To make things easier for the developer when creating many event handlers and thus save on some typing, the `Component` can be used and subclassed instead which provides an implicit mechanism for creating event handlers.

Basically every method in the component is automatically and implicitly marked as an event handler with `@handler(<name>)` where `<name>` is the name of each method applied.

The only exceptions are:

- Methods are start with an underscore `_`.
- Methods already marked explicitly with the `@handler(...)` decorator.

---

**Note:** You can specify that a method not be an event handler by marking it with `@handler(False)`.

---

## Values

## Debugging

## Tools

## Manager

### 8.1.4 The circuits.web Framework

#### 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:

<b>Events</b>	Event	Description
	Request	The Request Event
	Response	The Response Event
	Stream	The Stream Event

<b>Servers</b>	Server	Description
	BaseServer	The Base Server (no Dispatcher)
	Server	The <b>full</b> Server + Dispatcher

<b>Error Events</b>	Error	Description
	HTTPError	A generic HTTP Error Event
	Forbidden	A Forbidden (403) Event
	NotFound	A Not Found (404) Event
	Redirect	A Redirect (30x) Event

<b>Dispatchers</b>	Dispatcher	Description
	Static	A Static File Dispatcher
	Dispatcher	The Default Dispatcher
	VirtualHosts	Virtual Hosts Dispatcher
	XMLRPC	XML-RPC Dispatcher
	JSONRPC	JSON-RPC Dispatcher

<b>Other Components</b>	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:

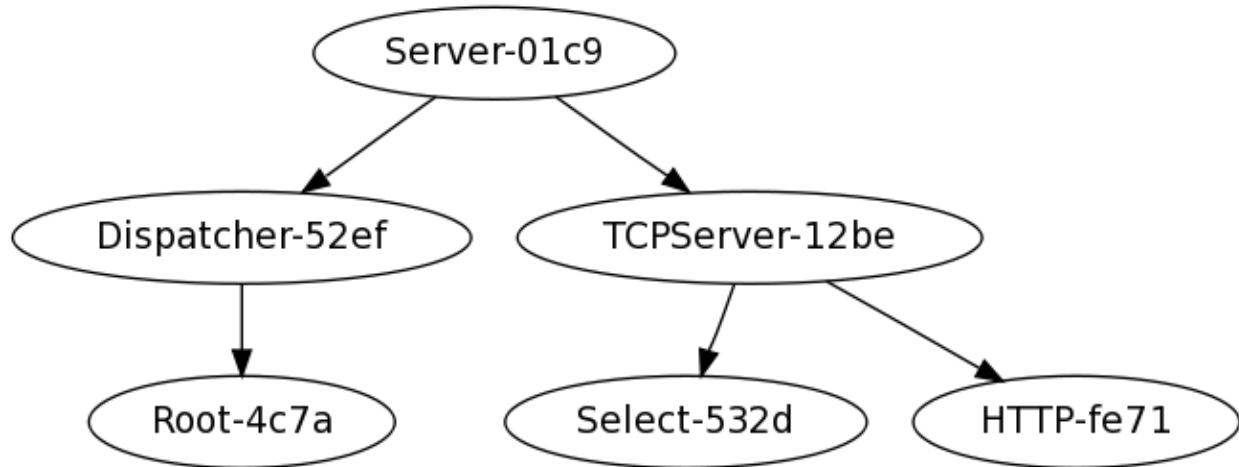
```

1 from circuits import Component
2 from circuits.web import BaseServer
3
4 class Root(Component):
5
6     def request(self, request, response):
7         return "Hello World!"
8
9 (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](#).

## 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 circuits.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:

```
1 from circuits.web import Server, Controller  
2  
3 class Root(Controller):  
4  
5     def index(self):  
6         return "Hello World!"  
7  
8 (Server(8000) + Root()).run()
```

Have fun!

## 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 `circuits.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.

## A Stand Alone Server

A stand alone server consist of the components shown in section [Getting Started](#). The process of handling an HTTP request starts with the `TCPServer` receiving a chunk of bytes. It emits those bytes as a `Read` event on the channel shared by the `Server`, `HTTP`, `TCPServer` and `Dispatcher` components (“web” by default).

The `Read` events are handled by the `HTTP` component. It collects the chunks until a complete HTTP request has been received. The request is then emitted as a `Request` event with an instance of classes `Request` and `Response` each as arguments. To complete the client’s request, a `Response` event must be fired. This is usually done by the `HTTP` component itself upon the receipt of a `RequestSuccess` event (automatically generated after all handlers for the `Request` event have been invoked successfully). In case of a problem, the `Request` event’s handlers should fire or return a `HTTPError` which is instead converted by the `HTTP` component to a `Response` event.

`HTTP`’s handler for the `Response` event retrieves the response information from the event and encodes it as required by `HTTP` (the protocol). It then fires one or more `Write` events which are handled by the `TCPServer` (and the response is thus sent to the client). More details can be found in [circuits.web.http – HTTP Protocol](#).

A commonly used component for handling `Request` events is a dispatcher. [To be continued]

## 8.1.5 HowTos

### 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 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

```

1  #!/usr/bin/env python
2
3  from circuits import Component
4  from circuits.net.sockets import TCPServer, Write
5
6  class Server(Component):
7
8      def __init__(self, host, port=8000):
9          super(Server, self).__init__()
10
11         self._clients = []
12
13         TCPServer((host, port)).register(self)
14
15     def connect(self, sock, host, port):
16         self._clients.append(sock)
17
18     def disconnect(self, sock):
19         self._clients.remove(sock)
20
21     def read(self, sock, data):
22         for client in self._clients:
23             if not client == sock:
24                 self.fire(Write(client, data.strip()))
25
26 Server("localhost").run()
```

Download `simple_server.py`

## 8.1.6 API Reference

### 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.

**classmethod handlers** ()

Returns a list of all event handlers for this `Component`

**classmethod handles** (*\*names*)

Returns True if all names are event handlers of this `Component`

**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 uncamed 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.

```
class circuits.core.events.GenerateEvents(lock, max_wait)
Bases: circuits.core.events.Event
```

Generate events event

This event is sent by the circuits core. All components that generate timed events or events from external sources (e.g. data becoming available) should fire any pending events in their “generate\_events” handler. The handler must either be a filter (preventing other handler from being called in the same iteration) or must invoke `reduce_time_left()` with parameter 0.

**Parameters** **max\_wait** – maximum time available for generating events.

Components that actually consume time waiting for events to be generated, thus suspending normal execution, must provide a method `resume` that interrupts waiting for events.

#### **time\_left**

The time left for generating events. A value less than 0 indicates unlimited time. You should have only

one component in your system (usually a poller component) that spends up to “time left” until it generates an event.

**reduce\_time\_left** (*time\_left*)

Update the time left for generating events. This is typically used by event generators that currently don’t want to generate an event but know that they will within a certain time. By reducing the time left, they make sure that they are reinvoked when the time for generating the event has come (at the latest).

This method can only be used to reduce the time left. If the parameter is larger than the current value of time left, it is ignored.

If the time left is reduced to 0 and the event is currently being handled, the handler’s *resume* method is invoked.

**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.

### 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 its 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.handlers.reprhandler(handler)
```

### `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 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

**pid**

Return the process id 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*, *link=None*)

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 causes an invocation of `run()` to return.

**tick(timeout=-1)**

Execute all possible actions once. Process all registered 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(socket=None)**

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)`

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.

### Variables

- **result** – True if this value has been changed.
- **errors** – True if while setting this value an exception occured.
- **notify** – True or an event name to notify of changes to this value

This is a `Future/Promise` implementation.

**Components** none

Events none

Functions none

#### `circuits.core.pollers` – I/O Pollers

Poller Components for asynchronous file and socket I/O.

This module contains Poller components that enable polling of file or socket descriptors for read/write events. Pollers:  
- Select - Poll - EPoll

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.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` (*\*args*, *\*\*kwargs*)

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)

initializes x; see `x.__class__.__doc__` for signature

#### 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**

#### `circuits.io` – **I/O Support**

I/O Support

This package contains various I/O Components. Provided are 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 uncamed 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 uncamed 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 uncamed 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 uncamed 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 uncamed 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 uncamed 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 uncamed 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 uncamed 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(*args, **kwargs)`  
 Bases: `circuits.core.components.Component`  
 initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**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

## `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 uncamed 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 uncamed 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 the 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 appropriate event. This must not be overridden by sub-classes, if it is, this must be explicitly 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 uncamed 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 tuple 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 whose 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 uncamed 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 uncamed 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 the 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 appropriate event. This must not be overridden by sub-classes, if it is, this must be explicitly 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 sent 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 uncamed 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 tuple 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 whose 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.SocketError` (\*args)

Bases: `circuits.core.events.Event`

SocketError 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.BaseComponent`

**channel** = 'client'

**parse\_bind\_parameter** (*bind\_parameter*)

**connected**

**close** ()

**write** (*data*)

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

Bases: `circuits.net.sockets.Client`

**socket\_family** = 2

**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, chan-
    nel='server', **kwargs)
    Bases: circuits.core.components.BaseComponent
    channel = 'server'
    parse_bind_parameter (bind_parameter)
    connected
    host
    port
    close (sock=None)
    write (sock, data)

class circuits.net.sockets.TCPServer (bind, secure=False, backlog=5000, bufsize=4096, chan-
    nel='server', **kwargs)
    Bases: circuits.net.sockets.Server
    socket_family = 2
    parse_bind_parameter (bind_parameter)

class circuits.net.sockets.UDPServer (bind, secure=False, backlog=5000, bufsize=4096, chan-
    nel='server', **kwargs)
    Bases: circuits.net.sockets.Server
    socket_family = 2
    close ()
    write (address, data)
    broadcast (data, port)

class circuits.net.sockets.UNIXServer (bind, secure=False, backlog=5000, bufsize=4096, chan-
    nel='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.
```

## 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 uncamed 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)
```

**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 structure. @rtype: str

**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** (*host=None, port=None, secure=None*)

**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 uncamed 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, de-
                                                faults=('index.html', 'index.shtml'), dirlist-
                                                ing=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 another 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**

## Events

## Components

**class** `circuits.web.dispatchers.websockets.WebSockets` (*path=None*, *wchannel='ws'*, *\*args*, *\*\*kwargs*)

Bases: `circuits.core.components.BaseComponent`

This class implements an RFC 6455 compliant WebSockets dispatcher that handles the WebSockets handshake and upgrades the connection.

The dispatcher listens on its channel for `Request` events and tries to match them with a given path. Upon a match, the request is checked for the proper Opening Handshake information. If successful, the dispatcher confirms the establishment of the connection to the client. Any subsequent data from the client is handled as a WebSocket data frame, decoded and fired as a `Read` event on the *wchannel* passed to the constructor. The data from `Write` events on that channel is encoded as data frames and forwarded to the client.

Firing a `Close` event on the *wchannel* closes the connection in an orderly fashion (i.e. as specified by the WebSocket protocol).

### Parameters

- **path** – the path to handle. Requests that start with this path are considered to be WebSocket Opening Handshakes.
- **wchannel** – the channel on which `Read` events from the client will be delivered and where `Write` events to the client will be sent to.

## 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 uncamed 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.

**Components** none**Events**

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

Bases: `circuits.core.events.Event`

An event for signaling an HTTP error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

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

Bases: `circuits.web.errors.HTTPError`

An event for signaling the HTTP Forbidden error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

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

Bases: `circuits.web.errors.HTTPError`

An event for signaling the HTTP Not Found error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

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

Bases: `circuits.web.errors.HTTPError`

An event for signaling the HTTP Redirect response

The constructor creates a new instance and modifies the *response* argument to reflect a redirect response to the given *url*.

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

Bases: `circuits.web.errors.HTTPError`

An event for signaling the HTTP Unauthorized error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**Functions** none**circuits.web.events – Events****Events**

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

Bases: `circuits.core.events.Event`

WebEvents have both their *success* and *failure* attributes set to True. So event processing generates the derived events ...Success or ...Failure events.

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 uncamed 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 uncamed 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 uncamed 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 uncamed 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, trace-*  
*back=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 server side Hyper Text Transfer Protocol or commonly known as HTTP.

#### 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.

The component handles `Read` events on its channel and collects the associated data until a complete HTTP request has been received. It parses the request's content and puts it in a `Request` object and creates a corresponding `Response` object. Then it emits a `Request` event with these objects as arguments.

The component defines several handlers that send a response back to the client.

**`_on_response`** (*response*)

Response Event Handler

**Parameters** `response` (`Response`) – the `Response` object created when the HTTP request was initially received.

This handler builds an HTTP response data stream from the information contained in the `response` object and sends it to the client (firing `Write` events).

**`_on_read`** (*sock, data*)

Read Event Handler

Process any incoming data appending it to an internal buffer. Split the buffer by the standard HTTP delimiter CRLF and create `Raw` Event per line. Any unfinished lines of text, leave in the buffer.

**`_on_httperror`** (*event, request, response, code, \*\*kwargs*)

Default HTTP Error Handler

Default Error Handler that by default just fires a `Response` event with the `response` as argument. The `response` is normally modified by a `HTTPError` instance or a subclass thereof.

**`_on_request_success`** (*e, value*)

Handler for the `RequestSuccess` event that is automatically generated after all handlers for a `Request` event have been invoked successfully.

#### Parameters

- **`e`** – the successfully handled `Request` event (having as attributes the associated `Request` and `Response` objects).
- **`value`** – the value(s) returned by the invoked handler(s).

This handler converts the value(s) returned by the (successfully invoked) handlers for the initial `Request` event to a body and assigns it to the `Response` object's `body` attribute. It then fires a `Response` event with the `Response` object as argument.

**Events** See `circuits.web.events.Request` in *circuits.web.events – Events*.

**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

`circuits.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', secure=False, certfile=None, 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 Controllers 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.websocket` – WebSocket Client

### Events

### Components

**class** `circuits.web.websocket.WebSocketClient` (*url*, *channel='wsclient'*, *wchannel='ws'*, *headers={}*)

Bases: `circuits.core.components.BaseComponent`

An RFC 6455 compliant WebSocket client component. Upon receiving a `circuits.web.client.Connect` event, the component tries to establish the connection to the server in a two stage process. First, a `circuits.net.sockets.Connect` event is sent to a child `TCPClient`. When the TCP connection has been established, the HTTP request for opening the WebSocket is sent to the server. A failure in this setup process is signaled by a `NotConnected` event.

When the server accepts the request, the WebSocket connection is established and can be used very much like an ordinary socket by handling `Read` events on and sending `Write` events to the channel specified as the `wchannel` parameter of the constructor. Firing a `Close` event on that channel closes the connection in an orderly fashion (i.e. as specified by the WebSocket protocol).

### Parameters

- **url** – the URL to connect to.
- **channel** – the channel used by this component
- **wchannel** – the channel used for the actual WebSocket communication (read, write, close events)
- **headers** – additional headers to be passed with the WebSocket setup HTTP request

```

channel = 'wsclient'

close()

connected

```

## Functions

### 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 requested method.
- **scheme** (*str*) – The requested scheme.
- **path** (*str*) – The requested path.
- **protocol** (*str*) – The requested 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` (*\*args*, *\*\*kwargs*)  
Bases: `circuits.core.components.BaseComponent`  
initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature  
**class** `circuits.web.wsgi.Gateway` (*\*args*, *\*\*kwargs*)  
Bases: `circuits.core.components.BaseComponent`  
initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**Functions**   none

## `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 uncamed 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 uncamed 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 uncamed 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 it should retrieve 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 sent 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 uncamed 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 uncamed 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 uncamed 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 uncamed 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

### 8.1.7 Developer Docs

So, you'd like to contribute to circuits in 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.

## Development Introduction

Here's how we do things in circuits...

### Standards

We use the following coding standard:

- [pep8](#)

### Tools

We use the following tools to develop circuits and share code:

- **Code Sharing:** [Mercurial](#)
- **Code Hosting and Bug Reporting:** [BitBucket](#)
- **Project Management:** [Pivotal Tracker](#)
- **Documentation Hosting:** [Read the Docs](#)
- **Package Hosting:** [Python Package Index \(PyPi\)](#)

## Contributing to circuits

Here's how you can contribute to circuits

### Share your story

One of the best ways you can contribute to circuits is by using circuits. Share with us your story of how you've used circuits to solve a problem or create a new software solution using the circuits framework and library of components. See our [Users Page](#).

### Submitting Bug Reports

We welcome all bug reports. We do however prefer bug reports in a clear and concise form with repeatable steps. One of the best ways you can report a bug to us is by writing a unit test (//similar to the ones in our tests//) so that we can verify the bug, fix it and commit the fix along with the test.

**To submit a bug report, please use:** <http://bitbucket.org/prologic/circuits/issues>

### Writing new tests

We're not perfect, and we're still writing more tests to ensure quality code. If you'd like to help, please [Fork circuits-dev](#), write more tests that cover more of our code base and submit a [Pull Request](#). Many Thanks!

### Adding New Features

If you'd like to see a new feature added to circuits, then we'd like to hear about it~ We would like to see some discussion around any new features as well as valid use-cases. To start the discussions off, please either:

- [Chat to us on #circuits on the FreeNode IRC Network](#)
- or
- [Submit a NEW Issue](#)

### Development Processes

We document all our internal development processes here so you know exactly how we work and what to expect. If you find any issues or problems please let us know!

#### Software Development Life Cycle (SDLC)

We employ the use of the [SCRUM Agile Process](#) and use [Pivotal Tracker](#) to track our stories, bugs, chores and releases. If you wish to contribute to circuits, please familiarize yourself with SCRUM and Pivotal Tracker.

### Bug Reports

- New Bug Reports are submitted via: <http://bitbucket.org/prologic/circuits/issues>
- Confirmation and Discussion of all New Bug Reports.
- Once confirmed, a new Bug is raised in [Pivotal Tracker](#) into the IceBox.

### Feature Requests

- New Feature Requests are submitted via: <http://bitbucket.org/prologic/circuits/issues>
- Confirmation and Discussion of all New Feature Requests.
- Once confirmed, a new Feature is raised in [Pivotal Tracker](#) into the IceBox.

### Writing new Code

- Write your code.
- Use [flake8](#) to ensure code quality.
- Run the tests:

```
$ tox
```
- Ensure any new or modified code does not break existing unit tests.
- Updated any relevant docstrings or documentatino.

## Running the Tests

To run the tests you will need the following installed:

- `tox` installed as well as
- `pytest-cov`
- `pytest`

All of these can be installed via `easy_install` or `pip`.

Please also ensure that you have all supported versions of Python that circuits supports installed in your local environment.

To run the tests:

```
$ tox
```

## 8.1.8 ChangeLog

### circuits-2.1.0 20130224 (<release>)

#### Features

- Added IPv6 Support [#39987887]
- Document and Comment all Examples [#40112459]
- Update WebSockets dispatcher in `circuits.web` [40064247]
- Component Event Handlers Property [#41573841]
- Component targeting performance [#41644031]
- Multi-App WSGI Gateway [#40071339]
- More efficient timer implementation [#43700669]

#### Bugs

- Fix the broken links in the Users page [#40063597]
- `circuits.tools.inspect` shows incorrect ticks [#41650961]
- Race Condition with `call/wait` primitives [#41643711]
- RESTful Dispatcher [#41643773]
- Fix Component instance targeting [#41644025]
- Worker Processes [#41643831]
- Make `examples/web/sslserver.py` work again [42007509]
- Value instance in body [43287263]

## Chores

- Import shadowing in `manager.py` [#41644023]
- Run the codebase through `flake8` [#41651093]
- URL Quoting Unit Test [#41882655]
- SSL Unit Test [#42849393]
- Python 3.x Support [43027337]
- Fill out FAQ in Docs [#43389199]
- Write Developer Docs [#43389193]

## Other Changes

A list of other changes we forgot to track :)

- Fixed latency timing for `circuits.bench` on win32
- Updated the URL for ReadTheDocs
- Updated the Getting Started / Downloading documentation to say something about the Tags tab for downloading archives.
- Removed unused import in `circuits.app.env`
- Added some documentation about how HTTP requests are handled.
- Removed unused code
- A simple chat server implementation (Examples)
- Fixed a bug with `Manager.processTasks` and Error event firing. handler was not being set correctly
- Fixes wrong variable names in `sockets.py`
- Fixed an `UnboundLocalError` bug
- Expect the original method to be passed as input to `Manager.removeHandler` `Manager.addHandler` now returns a reference to the original method.
- Guard against a `None _sock` or non-existent `_sock` object.
- Allow a custom host, port and secure option to be passed to the connect event handler of `circuits.web.client.Client`
- Redesigned `tests/web/conftest.py` with newer `pytest` APIs using `@pytest.fixture(...)`
- Experimental new `Watcher/Manager` fixtures
- Docs for `Fallback` generator
- Removing the concept of `_ticks`, `@tick` and friends
- Fixes timers
- Reimplemented `circuits.io.File` as per `circuits.net.sockets`
- Re-implemented `Notify Component` and fixed test
- Reworked `Notifier Component` to be more `async` friendly
- Fixed a bug with the repr of `Manager` and thus `Components` with non-str channels. eg: `(1, 1)`

- Added a unit test for generate\_events handler
  - Fixed terminal example demonstrating Streaming with circuits.web
  - Fixed index page for terminal web example
  - Added an Open event to File
  - Fixed a bug with the static dispatcher for circuits.web whereby the path was not being unquoted
  - Fixed a bug with the static circuits.web dispatcher whereby directory listing did not quote urls
  - Added simple examples demonstrating circuits' primitives
  - Added example of .fire(...) primitive
  - Restructured circuits.io and implemented circuits.io.Process with unit test
  - Added Hello World example
  - Made components started in process mode not link by default. Have to explicitly set the kwarg `link=component_instance`.
  - Fixed raising of `SystemExit` or `KeyboardInterrupt`.
  - Modified the way caching works with the Static dispatcher
  - Brought back the `circuits.core.bridge.Bridge`. With it better inter-process communications via high-speed full duplex pipes using `circuits.net.sockets.Pipe` (UNIX Only), a much simpler API.
- In addition, a change to the way Value Notification works. `.notify` of an event instance can either be `True` or a channel to send the `ValueChanged` event to.

- Added timers examples resurrected from old circuits
- Global handlers and component targeting Global handlers would be ignored when using component targeting.

This patch considers them. To do this, we have added an extra property to managers.

You can use `traverse_children_handlers` to increase performance when you have a huge number of components. Put the components that are only meant to be used with component targeting under a single component. The hierarchy looks like this:

```
Root -> Umbrella Component -> Component 1, Component 2, Component 3, ...
    -> Debugger()
    -> etc
```

- Set Umbrella Component `traverse_children_handlers` to false to prevent traversing the huge number of children components.
- Fixed Connection header interpretation.
- Updated documentation for WebSocket.
- Removed pool - covered by worker
- Fixed dispatchers return value.
- Firing Connect event when a web socket connection is established to make behavior look even more like ordinary sockets.
- Nuked `@future` decorator due to pickling problems for processes.
- Allow coroutine-style tasks to terminate the system via raise of `SystemExit` or `KeyboardInterrupt`
- Dropping events unnoticed when waiting is definitely a bad idea. Fixed.
- Clarification on implementing a `GenerateEvents` handler.

- Optimized GenerateEvents handling.
- Optimized management of FallbackGenerator.
- Fixed a problem with events being out of sequence when `_fire` is called recursively. The fix exposes a problem with `confest.Waiter` (events getting lost because they are produced too fast, therefore queue is increased). Reducing production rate will be in next commit.
- Small fix for new event queue handling.
- Fixed problem with handler cache update and concurrent event handling/structure changes. This happens e.g. in unit tests when the app is started and the test program adds or removes components concurrently.
- Optimized/clarified GenerateEvents implementation.
- One more concurrency problem fixed.
- Fixed generate events handler.
- Fixed bug with handler cache being always invalidated. Avoid `stop()` acting as concurrent thread on `_dispatcher()`
- Fixed payload length calculation in web sockets protocol.
- Some very small - but measurable - performance improvements. Checking them in mainly because now no one else needs to think about whether they have an effect.
- Fixed IPv6 socket tests for OSX and badly configured IPv6 networks
- Fixed ready handler for `test_node`
- Re-added an old revived example of an IRC Client integrating the `urwid curses` library for the interface
- Added an example of using `yield` to perform cooperative multi-tasking inside a request event handler
- Uses `echo` for `test_process`
- Fixes process stdout/stderr handling Process was not waiting for all output from a process to have been processed and resulted sometimes some of the process output being lost while stopping.
- Added a fall back error handler, so problems aren't discarded silently any more.
- Fixed a `TypeError` being raised in the request handler for `WebSockets` dispatcher
- Prevent the underlying `TCPClient` connect handler from inadvertently being triggered by connect events being fired to the web client
- Added `tox` configuration file. Just run: `tox`
- Configure `tox` to output `junitxml`
- Fixed the logic of `path` -> `wsgi` dispatching in `Gateway`
- Fixed an awful bug with `wrappers.Response` whereby a default `Content-Type` was always set regardless and didn't allow anything else to set this.
- Fixed `test_disps` so it doesn't use an existing port that's in use.
- Added a missing piece of `WSGI 1.0` functionality for `wsgi.Gateway`. The `write()` callable
- Rewrite `test_filer` unit test. Ensured `EOF` event is not triggered for files opened in `a` or `+` modes.
- Added simple examples demonstrating circuits' primitives
- Added example of `.fire(...)` primitive
- Restructured `circuits.io` and implemented `circuits.io.Process` with unit test
- Forgot to add `test_process` unit test



- Added Hello World example
- Fixed event handler for `generate_events`
- Implemented multiprocessing support with communication via full duplex pipes. Fixed worker processes and added unit test
- Made components started in process mode not link by default. Have to explicitly set the kwarg `link=True`
- Fixed process pools and added unit test
- Fixed name of method for handling inotify events - conflicting with `._process` of `Manager/Component`
- Renamed `._process` to `.__process` so as to not conflict with `Worker`
- Fixed raising of `SystemExit` or `KeyboardInterrupt`
- Trying to fix sending/receiving of events and their values
- Fixed IPv6 address evaluation.
- WebSockets reimplemented as specified by RFC 6455.
- Modified the way caching works with the `Statis` dispatcher and `serve_file(...)` function. Set Last-Modified instead of a large Expires header.
- Marked as a Missing Feature - still being worked on.
- Brought back the `circuits.core.bridge.Bridge`. With it better inter-process communications via high-speed full duplex pipes using `circuits.net.sockets.Pipe` (UNIX Only), a much simpler API.  
In addition, a change to the way Value Notification works. `.notify` of an event instance can either be `True` or a channel to send the `ValueChanged` event to.
- Causes errors. Wrong thing to do here.
- Use `uuid4` for worker channels.
- Removed left over debugging code
- Added capability of components started in process mode to link to arbitrary managers other than themselves (if `link=True`)
- Added unit test for testing a component in process mode linked with another parent system separately
- Added timers examples resurrected from old circuits
- Thread safety for watcher
- Reverts `test_component_targeting.py`
- Fixed Connection header interpretation.
- Updated documentation for `WebSocket`.
- Changed the way process linking works by not starting the parent in thread mode.
- Fixes watcher test fixture
- Implemented `.handlers()` and re-implemented `.handles()`. [Delivers #41573841]
- Removed superfluous bridge test
- Fixed usage of `.start` starting Workers in process mode
- Better internal variable name
- Start worked either when the system starts or we are registered to an already started system
- Implemented `.handlers()` correctly

- Set unique channels for the Pipe and Bridge when bridging processes
- Removed. Going to reimplement all this
- Don't cause a nasty fork bomb!
- Marked test\_process\_pool as skipping XXX: Broken Functionality
- Fixed sleeping when nothing to do adding needs\_resume logic to the pollers
- Proposed new Worker Component – wrapping multiprocessing.Process
- Accidentla commit
- Trying to re-implement Pool...
- Marked some tests as broken
- Added support for both Process and Thread in Worker component with same API.
- Reverted changes to the task handler for Worker
- Removed .resume()
- Fixed some brokenness in future
- Trying to make @future work with Worker in process mode
- Lock bench
- Switches worker to apply\_async
- Allow the poller to sleep when tasks are running
- Use ThreadPool for worker threads
- Cleaned up Worker
- benchmark only branch
- Removed pool - covered by worker
- Workaround for Threadpool bug started from secondary threads
- Fixed dispatchers return value.
- Firing Connect event when a web socket connection is established to make behavior look even more like ordinary sockets.
- Restored 2.0.1 GenerateEvents handling with minor enhancements to get things working again.
- Nuked @future decotator due to pickling problems for processes.
- Unmarked test as failing. Test now passes prefectly
- Allow coroutine-style tasks to terminate the system via raise of SystemExit or KeyboardInterrupt
- Fixed SSL support for circuits.web [Delivers #42007509]
- Dropping events unnoticed when waiting is definitely a bad idea. Fixed.
- Clarification on implementing a GenerateEvents handler.
- Added missing import (timed wait with FallBackGenerator cannot have worked for some time).
- Optimized GenerateEvents handling.
- Backed out of changeset 3413:98d0056ef18a
- Optimized management of FallbackGenerator.

- Fixed a problem with events being out of sequence when `_fire` is called recursively. The fix exposes a problem with `conftest.Waiter` (events getting lost because they are produced too fast, therefore queue is increased). Reducing production rate will be in next commit.
- Small fix for new event queue handling.
- Fixed problem with handler cache update and concurrent event handling/structure changes. This happens e.g. in unit tests when the app is started and the test program adds or removes components concurrently.
- Optimized/clarified `GenerateEvents` implementation.
- One more concurrency problem fixed.
- Fixed generate events handler.
- Fixed bug with handler cache being always invalidated. Avoid `stop()` acting as concurrent thread on `_dispatcher()`.
- Added unit test for url quoting of static files for the static `circuits.web` dispatcher. [Delivers #41882655]
- Added unit test for secure `circuits.web` server. [Delivers #42849393]
- Fixed creation/registration/location of poller upon startup or registration
- Skip `test_secure_server` if `ssl` is not available
- Fixed payload length calculation in web sockets protocol.
- Some very small - but measurable - performance improvements. Checking them in mainly because now no one else needs to think about whether they have an effect.
- Fixed conflicting attribute.
- Added new `.pid` property to `Manager` fixing `tests.core.test_bridge` - Broken by 70677b69bf05
- Fixed `ipv6` socket tests for OSX and badly configured `IPv6` networks
- Ugly fix for this test
- Fixed ready handler for `test_node`
- For some reason removing `max_events` in the `Watcher` fixture for esting purposes fixes some tests on OSX
- Small sample to hopefully test for memory leaks :)
- Improved `.handlers()` to recursively return handlers of it's subclasses whilst filtering out private handlers (`_.*`)
- Use `BaseComponent` for convenience for non-handler methods
- Return a list of all handlers except handlers listneing to private events
- Left over print - oops :)
- Moved to `examples/`
- Re-added an old revived example of an IRC Client integrating the `urwid` curses library for the interface
- Improved comment.
- Added multi-add support for Gateway web compnoent [Delivers #40071339]
- Fixed Python 3 compatibility for retrieving the `.resumse()` method. Fixes Issue #35
- Removed unused `StringIO` import
- A bunch of Python 3.1 compatibility fixes (mostly import fixes)
- Added an example of using `yield` to perform cooperative multi-tasking inside a request event handler
- Uses `echo` for `test_process`

- Fixes process stdout/stderr handling Process was not waiting for all output from a process to have been processed and resulted sometimes some of the process output being lost while stopping.
- A bunch more Python 3 fixes. Using the six module here.
- Added a fall back error handler, so problems aren't discarded silently any more.
- Ooops accidentally committed this
- Fixed some Python 3 import issues
- More Python 3 fixes
- Fixed Python 3 issue with iterators
- Use range for Python 3 compatibility
- Fixed assertions for Python 3 compat
- Accidentally committed a test with Debugger
- Fixed value\_changed handler to check correctly for binary types (Python 3 fixes)
- Python 3 fixes for string type checks
- Refactored for Python 3
- More Python 3 fixes. Marked the rest as Broken on Python 3
- Fixed broken web tests for Python 3 by wrapping the request body in a TextIOWrapper for FieldStorage
- Fixed XML and JSON RPC Dispatchers for Python 3
- Replace .has\_key() with in for Python 3 compatibility
- Fixed a TypeError being raised in the request handler for websockets dispatcher
- Prevent the underlying TCPClient connect handler from inadvertently being triggered by connec events being fired to the web client
- Unmarked as skipping. No longer broken on Python 3
- Finished cleaning up my code base.
- Removed some debugging junk I had forgotten to get rid of.
- Fixed File for Python 3 support adding optional encoding support
- Fixed Process for Python 3
- Ooops :/
- Fixed test\_logger\_Error for Python 3
- Fixed Bridge for Python 3
- Fixed Node for Python 3
- Added pytest.PYVER and Marked test\_secure\_server web test as Broken on Python 3.2 (needs fixing)
- Marked a bunch of 3.2 and 3.3 specific tests that are broken with these versions (needs looking into)
- Removed Broken on Python 3 marked - these tests now pass on 3.1 3.2 and 3.3
- Fixed SSL Sockets for Python 3.2/3.3 (Should do\_handshake() be executed in a thread?)
- Added tox configuration file. Just run: run
- Configure tox to output junitxml
- Fixed tox configuration

- Assuming localhost was incorrect. Sorry Mark :/
- Fixed test\_logger for pypy
- Backed out changeset 7be64d8b6f7c
- Reconfigured tox settings
- Hopefully fixed tox.ini to detect jenkins
- Fixed tox.ini to work on jenkins (can't seem to auto detect jenkins :/)
- Added tox config for checking documetnation
- Changed the output filename of xml resutls for docs env (tox)
- Added pytest-cov as a dep to the docs env (tox)
- Configured coverage to output xml output
- Removed ptest-cov dep from docs env (tox)
- Trying to fix the randomly failing test\_node test.
- Oops broke wait\_for - fixed hopefully
- Backed out changeset 795712654602
- Backed out changeset 1ee04d5fb657
- Added ignore for generated junitxml output files
- Hopefully an improved unit test for node using the manager and watcher fixtures.
- Updated with supported version of Python
- Fixed the logic of path -> wsgi dispatching in Gateway
- Fixed an awful bug with wrappers.Response whereby a default Content-Type was always set regardless and didn't allow anything else to set this.
- Fixed test\_disps so it doesn't use an existing port that's in use.
- Added a missing piece of WSGI 1.0 functionality for wsgi.Gateway – The write() callable
- Write bytes in this test
- Correctly handle unicode (I think)
- Fixed a bug with null responses from WSGI Applications hosted by wsgi.Gateway. Empty class did not implement \_\_nonzero\_\_ for Python 2.x [Delivers #43287263]
- Remvoed pyinotify dep from tox config so Windows tests can run
- Skip test\_unixserver for Windows
- Skip this test on Windows
- Skip these tests on Windows
- Skip this test on Windows
- Fixed test\_tcp\_bind for Windows
- Updated docs and re-added FAQ from old wiki (updated slightly) [Delivers #43389199]
- Fixed bottom of FAQ
- Updated Change Log [#42945315]
- Updated Release Notes [#42945315]

- Fixed list of new changes for 2.1.0 release notes
- Updated Developet Docs
- Bumped version to 2.1.0
- More resource efficient timer implementation [Delivers #43700669].
- Fixed a problem with cLength being unknown if self.body == False.
- Test fails on shining panda py32 only. May be a race condition (wait\_for using same loop interval as timer interval). Checking in for testing.
- Fixed problem with “Content-Length: 0” header not being generated for empty responses.
- Backed out of changeset 3575:ece3ee5472ef, makes CI hang for unknown reason.
- Hopefully finally fixed problems with timer test on CI platform.
- Updated pypi classifier for circuits
- Fixed random problems with opening SSL socket.
- Fixed concurrence problem (doing reset() in event loop and calling unregister() concurrently).
- Modified test a bit to find out what happens in CI environment (problem not reproducible in local py32).
- Calling headers.setdefault twice for the same name doesn’t make sense.
- Adapted test case to previous fix.
- Fixed problem with “Content-Length: 0” header not being generated for empty responses.
- Fixed insecure usage of class variable.
- Reverted to old timer implementation. Cannot find bug in new version that only occurs in CI/py32 environment (cannot be reproduced locally).
- Make sure that check runs faster than timer increments.
- Added missing Content-Length header (must be present according to RFC 2616).
- Provide a more informative representation.
- Fixed a docstring typo
- Just re-raise the exception here rather than hide it with an InternalServer exception
- Potential failing test for [#43885227]
- Fixed problem with receiving incomplete header data (first chunk doesn’t already include empty line that separates headers and body).
- Fixed problem with header not being detected if a chunk ends exactly after rn of last header line and next chunk starts with empty line (rn) before body.
- Fixes inconsisnt semantic behavior in channel selection.
- Fixed expression for \_handler\_channel in Manager.getHandlers(...)
- Fixed (and simplified) handler\_channel evaluation.
- Fixed channel representation for cases where channel is a component.
- Adapted prepare\_unregister\_complete handling to fixed semantics for using managers as channels.
- Notifications are to be signaled to the given manager. Now that using managers as channels works properly, it showed that hey never were.
- Value change notifications are fired using Value’s manager as channel, so mke sure to listen on that.

- Value's notifications are fired on its manager component – by default the component that fired the event the value belongs to. Moved handler to proper place (is better place anyway).
- Encoding is necessary for this to succeed on py3.2 (urllib2 there doesn't accept body of type str).
- Added missing files to manifest. [Delivers #44650895] (Closes Issue #36)
- Fixing encoding problems.
- Fixing py2/py3 problem by using bytearray as obviously the only common denominator.
- multipart/form-data is binary. Boundaries are ASCII, but data between boundaries may be anything (depending on part's header)
- Trying to fix unicode issues
- Marked test\_node as Broken on Windows - Pickling Error with thread.lock
- Trying to fix broken unit test on Windows
- Setup docs with Google Analytics
- Marked test\_tcp\_connect\_closed\_port(...) as Broken on Windows – Need to fix this; this test is a bit weird :/
- Marked test\_tcp\_reconnect(...) test as Broken on Windows
- Updated state of test\_tcp\_reconnect(...) test on Windows – Apparently only Broken on Windows on Python 3.2 (odd)
- Fixed TypeError
- Marked as Broken on pypy – For some reason we're getting x00 (null bytes) in the stream when using the Python std lib logger.
- Marked tests.core.test\_debugger.test\_filename(...) as Broken on pypy – Need to investigate this
- Updated support platforms to include pypy

### **circuits-2.0.1 20121124**

- Fixed tests/web/test\_main.py which was badly written.
- Fixed a regression test testing the Debugger component

### **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 allowing 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](http://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 order
- Fixed missing import of `sys` in `circuits.io`

### 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` module 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.ERCONNREFUSED`
- 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

### 8.1.9 Users

#### Applications written in circuits

- [SahrisWiki](#) a Wiki / CMS / Blogging Engine.
- [kdb](#) a pluggable IRC Bot Framework.

#### 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 written by James Mills and Michael Anton.
- [Mangahelpers](#) Profiler for checking our webpages generation times, SQL and cache queries times, project page on <<https://launchpad.net/website-profiler>>

#### Other Users

- [MIT Media Lab](#), Uses circuits for several internal prototypes

### 8.1.10 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.

## 8.1.11 Frequently Asked Questions

### General

- ... **What is circuits?** circuits is an event-driven framework with a high focus on Component architectures making your life as a software developer much easier. circuits allows you to write maintainable and scalable systems easily
- ... **Can I write networking applications with circuits?** Yes absolutely. circuits comes with socket I/O components for tcp, udp and unix sockets with asynchronous polling implementations for select, poll, epoll and kqueue.
- ... **Can I integrate circuits with a GUI library?** This is entirely possible. You will have to hook into the GUI's main loop.
- ... **What are the core concepts in circuits?** Components and Events. Components are maintainable reusable units of behavior that communicate with other components via a powerful message passing system.
- ... **How would you compare circuits to Twisted?** Others have said that circuits is very elegant in terms of it's usage. circuits' component architecture allows you to define clear interfaces between components while maintaining a high level of scalability and maintainability.
- ... **Can Components communicate with other processes?** Yes. circuits implements currently component bridging and nodes
- ... **What platforms does circuits support?** circuits currently supports Linux, FreeBSD, OSX and Windows and is currently continually tested against Linux and Windows against Python versions 2.6, 2.7, 3.1 and 3.2
- ... **Can circuits be used for concurrent or distributed programming?** Yes. We also have plans to build more distributed components into circuits making distributing computing with circuits very trivial.

Got more questions?

- Send an email to our [Mailing List](#).
- Talk to us online on the [#circuits IRC Channel](#)

## 8.1.12 Glossary

VCS Version Control System, what you use for versioning your source code

### 8.1.13 Users

#### Applications written in circuits

- [SahrisWiki](#) a Wiki / CMS / Blogging Engine.
- [kdb](#) a pluggable IRC Bot Framework.

#### 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 written by James Mills and Michael Anton.
- [Mangahelpers](#) Profiler for checking our webpages generation times, SQL and cache queries times, project page on <<https://launchpad.net/website-profiler>>

#### Other Users

- [MIT Media Lab](#), Uses circuits for several internal prototypes

## 8.2 Indices and tables

- *Index*
- *modindex*
- *search*
- *Glossary*

# PYTHON MODULE INDEX

## C

circuits.app, ??  
circuits.app.config, ??  
circuits.app.daemon, ??  
circuits.app.env, ??  
circuits.app.log, ??  
circuits.core, ??  
circuits.core.components, ??  
circuits.core.debugger, ??  
circuits.core.events, ??  
circuits.core.handlers, ??  
circuits.core.loader, ??  
circuits.core.manager, ??  
circuits.core.pollers, ??  
circuits.core.timers, ??  
circuits.core.utils, ??  
circuits.core.values, ??  
circuits.core.workers, ??  
circuits.io, ??  
circuits.io.events, ??  
circuits.io.file, ??  
circuits.io.notify, ??  
circuits.io.serial, ??  
circuits.net, ??  
circuits.net.protocols, ??  
circuits.net.protocols.http, ??  
circuits.net.protocols.irc, ??  
circuits.net.protocols.line, ??  
circuits.net.sockets, ??  
circuits.node, ??  
circuits.node.client, ??  
circuits.node.events, ??  
circuits.node.node, ??  
circuits.node.server, ??  
circuits.node.utils, ??  
circuits.tools, ??  
circuits.web, ??  
circuits.web.client, ??  
circuits.web.constants, ??  
circuits.web.controllers, ??  
circuits.web.dispatchers.dispatcher, ??  
circuits.web.dispatchers.jsonrpc, ??  
circuits.web.dispatchers.static, ??  
circuits.web.dispatchers.virtualhosts, ??  
circuits.web.dispatchers.websockets, ??  
circuits.web.dispatchers.xmlrpc, ??  
circuits.web.errors, ??  
circuits.web.events, ??  
circuits.web.exceptions, ??  
circuits.web.headers, ??  
circuits.web.http, ??  
circuits.web.loggers, ??  
circuits.web.main, ??  
circuits.web.servers, ??  
circuits.web.sessions, ??  
circuits.web.tools, ??  
circuits.web.utils, ??  
circuits.web.websocket, ??  
circuits.web.wrappers, ??  
circuits.web.wsgi, ??