The Asphalt Framework (core)

Release 4.4.4.post8

Mar 26, 2019

Contents

This is the core Asphalt library. If you're looking for documentation for some specific component project, you will the appropriate link from the project's Github page.

If you're a new user, it's a good idea to start from the tutorials. Pick a tutorial that suits your current level of knowledge.

CHAPTER 1

Tutorials

The following tutorials will help you get acquainted with Asphalt application development. It is expected that the reader have at least basic understanding of the Python language.

Code for all tutorials can be found in the `examples` directory in the source distribution or in the Github repository.

1.1 Tutorial 1: Getting your feet wet – a simple echo server and client

This tutorial will get you started with Asphalt development from the ground up. You will be learn how to build a simple network server that echoes back messages sent to it, along with a matching client application. It will however not yet touch more advanced concepts like using the `asphalt` command to run an application with a configuration file.

1.1.1 Prerequisites

Asphalt requires Python 3.5.2 or later. You will also need to have the `venv` module installed for your Python version of choice. It should come with most Python installations, but if it does not, you can usually install it with your operating system's package manager (`python3-venv` is a good guess).

1.1.2 Setting up the virtual environment

Now that you have your base tools installed, it's time to create a *virtual environment* (referred to as simply `virtualenv` later). Installing Python libraries in a virtual environment isolates them from other projects, which may require different versions of the same libraries.

Now, create a project directory and a virtualenv:

```
mkdir tutorial1
cd tutorial1
python3.5 -m venv tutorialenv
source tutorialenv/bin/activate
```

On Windows, the last line should be:

```
tutorialenv\Scripts\activate
```

The last command *activates* the virtualenv, meaning the shell will first look for commands in its `bin` directory (`Scripts` on Windows) before searching elsewhere. Also, Python will now only import third party libraries from the virtualenv and not anywhere else. To exit the virtualenv, you can use the `deactivate` command (but don't do that now!).

You can now proceed with installing Asphalt itself:

```
pip install asphalt
```

1.1.3 Creating the project structure

Every project should have a top level package, so create one now:

```
mkdir echo
touch echo/__init__.py
```

On Windows, the last line should be:

```
copy NUL echo\__init__.py
```

1.1.4 Creating the first component

Now, let's write some code! Create a file named `server.py` in the `echo` package directory:

```
from asphalt.core import Component, run_application


class ServerComponent(Component):
    async def start(self, ctx):
        print('Hello, world!')

if __name__ == '__main__':
    component = ServerComponent()
    run_application(component)
```

The `ServerComponent` class is the *root component* (and in this case, the only component) of this application. Its `start()` method is called by `run_application` when it has set up the event loop. Finally, the `if __name__ == '__main__':` block is not strictly necessary but is good, common practice that prevents `run_application()` from being called again if this module is ever imported from another module.

You can now try running the above application. With the project directory (`tutorial`) as your current directory, do:

```
python -m echo.server
```

This should print "Hello, world!" on the console. The event loop continues to run until you press Ctrl+C (Ctrl+Break on Windows).

1.1.5 Making the server listen for connections

The next step is to make the server actually accept incoming connections. For this purpose, the `asyncio.start_server()` function is a logical choice:

```
from asyncio import start_server

from asphalt.core import Component, run_application


async def client_connected(reader, writer):
    message = await reader.readline()
    writer.write(message)
    writer.close()
    print('Message from client:', message.decode().rstrip())


class ServerComponent(Component):
    async def start(self, ctx):
        await start_server(client_connected, 'localhost', 64100)

if __name__ == '__main__':
    component = ServerComponent()
    run_application(component)
```

Here, `asyncio.start_server()` is used to listen to incoming TCP connections on the `localhost` interface on port 64100. The port number is totally arbitrary and can be changed to any other legal value you want to use.

Whenever a new connection is established, the event loop launches `client_connected()` as a new `Task`. Tasks work much like green threads in that they're adjourned when waiting for something to happen and then resumed when the result is available. The main difference is that a coroutine running in a task needs to use the `await` statement (or `async for` or `async with`) to yield control back to the event loop. In `client_connected()`, the `await` on the first line will cause the task to be adjourned until a line of text has been read from the network socket.

The `client_connected()` function receives two arguments: a `StreamReader` and a `StreamWriter`. In the callback we read a line from the client, write it back to the client and then close the connection. To get at least some output from the application, the function was made to print the received message on the console (decoding it from `bytes` to `str` and stripping the trailing newline character first). In production applications, you will want to use the `logging` module for this instead.

If you have the `netcat` utility or similar, you can already test the server like this:

```
echo Hello | nc localhost 64100
```

This command, if available, should print "Hello" on the console, as echoed by the server.

1.1.6 Creating the client

No server is very useful without a client to access it, so we'll need to add a client module in this project. And to make things a bit more interesting, we'll make the client accept a message to be sent as a command line argument.

Create the file `client.py` file in the `echo` package directory as follows:

```
import sys
from asyncio import open_connection

from asphalt.core import CLIApplicationComponent, run_application
```

(continues on next page)

(continued from previous page)

```
class ClientComponent(CLIApplicationComponent):
    def __init__(self, message: str):
        super().__init__()
        self.message = message

    async def run(self, ctx):
        reader, writer = await open_connection('localhost', 64100)
        writer.write(self.message.encode() + b'\n')
        response = await reader.readline()
        writer.close()
        print('Server responded:', response.decode().rstrip())

if __name__ == '__main__':
    component = ClientComponent(sys.argv[1])
    run_application(component)
```

You may have noticed that `ClientComponent` inherits from `CLIApplicationComponent` instead of `Component` and that instead of overriding the `start()` method, `run()` is overridden instead. This is standard practice for Asphalt applications that just do one specific thing and then exit.

The script instantiates `ClientComponent` using the first command line argument as the `message` argument to the component's constructor. Doing this instead of directly accessing `sys.argv` from the `run()` method makes this component easier to test and allows you to specify the message in a configuration file (covered in the next tutorial).

When the client component runs, it grabs the message to be sent from the list of command line arguments (`sys.argv`), converts it from a unicode string to a bytestring and adds a newline character (so the server can use `readline()`). Then, it connects to `localhost` on port 64100 and sends the bytestring to the other end. Next, it reads a response line from the server, closes the connection and prints the (decoded) response. When the `run()` method returns, the application exits.

To send the "Hello" message to the server, run this in the project directory:

```
python -m echo.client Hello
```

1.1.7 Conclusion

This covers the basics of setting up a minimal Asphalt application. You've now learned to:

- Create a virtual environment to isolate your application's dependencies from other applications
- Create a package structure for your application
- Start your application using `run_application()`
- Use asyncio streams to create a basic client-server protocol

This tutorial only scratches the surface of what's possible with Asphalt, however. The *second tutorial* will build on the knowledge you gained here and teach you how to work with components, resources and configuration files to build more useful applications.

1.2 Tutorial 2: Something a little more practical – a web page change detector

Now that you've gone through the basics of creating an Asphalt application, it's time to expand your horizons a little. In this tutorial you will learn to use a container component to create a multi-component application and how to set up a configuration file for that.

The application you will build this time will periodically load a web page and see if it has changed since the last check. When changes are detected, it will then present the user with the computed differences between the old and the new versions.

1.2.1 Setting up the project structure

As in the previous tutorial, you will need a project directory and a virtual environment. Create a directory named `tutorial2` and make a new virtual environment inside it. Then activate it and use `pip` to install the `asphalt-mailer` and `aiohttp` libraries:

```
pip install asphalt-mailer aiohttp
```

This will also pull in the core Asphalt library as a dependency.

Next, create a package directory named `webnotifier` and a module named `app` (`app.py`). The code in the following sections should be put in the `app` module (unless explicitly stated otherwise).

1.2.2 Detecting changes in a web page

The first task is to set up a loop that periodically retrieves the web page. For that, you can adapt code from the aiohttp HTTP client tutorial:

```
import asyncio
import logging

import aiohttp
from asphalt.core import CLIApplicationComponent, run_application

logger = logging.getLogger(__name__)


class ApplicationComponent(CLIApplicationComponent):
    async def run(self, ctx):
        with aiohttp.ClientSession() as session:
            while True:
                async with session.get('http://imgur.com') as resp:
                    await resp.text()

                await asyncio.sleep(10)

if __name__ == '__main__':
    run_application(ApplicationComponent(), logging=logging.DEBUG)
```

Great, so now the code fetches the contents of `http://imgur.com` at 10 second intervals. But this isn't very useful yet – you need something that compares the old and new versions of the contents somehow. Furthermore, constantly loading the contents of a page exerts unnecessary strain on the hosting provider. We want our application to be as polite and efficient as reasonably possible.

To that end, you can use the `if-modified-since` header in the request. If the requests after the initial one specify the last modified date value in the request headers, the remote server will respond with a `304 Not Modified` if the contents have not changed since that moment.

So, modify the code as follows:

```
class ApplicationComponent(CLIApplicationComponent):
    async def run(self, ctx):
        last_modified = None
        with aiohttp.ClientSession() as session:
            while True:
                headers = {'if-modified-since': last_modified} if last_modified else
↪{}
                async with session.get('http://imgur.com', headers=headers) as resp:
                    logger.debug('Response status: %d', resp.status)
                    if resp.status == 200:
                        last_modified = resp.headers['date']
                        await resp.text()
                        logger.info('Contents changed')

                await asyncio.sleep(10)
```

The code here stores the `date` header from the first response and uses it in the `if-modified-since` header of the next request. A `200` response indicates that the web page has changed so the last modified date is updated and the contents are retrieved from the response. Some logging calls were also sprinkled in the code to give you an idea of what's happening.

1.2.3 Computing the changes between old and new versions

Now you have code that actually detects when the page has been modified between the requests. But it doesn't yet show *what* in its contents has changed. The next step will then be to use the standard library `difflib` module to calculate the difference between the contents and send it to the logger:

```
from difflib import unified_diff


class ApplicationComponent(CLIApplicationComponent):
    async def run(self, ctx):
        with aiohttp.ClientSession() as session:
            last_modified, old_lines = None, None
            while True:
                logger.debug('Fetching webpage')
                headers = {'if-modified-since': last_modified} if last_modified else
↪{}
                async with session.get('http://imgur.com', headers=headers) as resp:
                    logger.debug('Response status: %d', resp.status)
                    if resp.status == 200:
                        last_modified = resp.headers['date']
                        new_lines = (await resp.text()).split('\n')
                        if old_lines is not None and old_lines != new_lines:
                            difference = diff.make_file(old_lines, new_lines,␣
↪context=True)
                            logger.info('Contents changed:\n%s', difference)

                        old_lines = new_lines

                await asyncio.sleep(10)
```

This modified code now stores the old and new contents in different variables to enable them to be compared. The `.split('\n')` is needed because `unified_diff()` requires the input to be iterables of strings. Likewise, the `'\n'.join(...)` is necessary because the output is also an iterable of strings.

1.2.4 Sending changes via email

While an application that logs the changes on the console could be useful on its own, it'd be much better if it actually notified the user by means of some communication medium, wouldn't it? For this specific purpose you need the `asphalt-mailer` library you installed in the beginning. The next modification will send the HTML formatted differences to you by email.

But, you only have a single component in your app now. To use `asphalt-mailer`, you will need to add its component to your application somehow. Enter `ContainerComponent`. With that, you can create a hierarchy of components where the `mailer` component is a child component of your own container component.

And to make the the results look nicer in an email message, you can switch to using `difflib.HtmlDiff` to produce the delta output:

```
from difflib import HtmlDiff


class ApplicationComponent(CLIApplicationComponent):
    async def start(self, ctx):
        self.add_component(
            'mailer', backend='smtp', host='your.smtp.server.here',
            message_defaults={'sender': 'your@email.here', 'to': 'your@email.here'})
        await super().start(ctx)

    async def run(self, ctx):
        with aiohttp.ClientSession() as session:
            last_modified, old_lines = None, None
            diff = HtmlDiff()
            while True:
                logger.debug('Fetching webpage')
                headers = {'if-modified-since': last_modified} if last_modified else
↪{}
                async with session.get('http://imgur.com', headers=headers) as resp:
                    logger.debug('Response status: %d', resp.status)
                    if resp.status == 200:
                        last_modified = resp.headers['date']
                        new_lines = (await resp.text()).split('\n')
                        if old_lines is not None and old_lines != new_lines:
                            difference = diff.make_file(old_lines, new_lines,␣
↪context=True)
                            await ctx.mailer.create_and_deliver(
                                subject='Change detected in %s' % event.source.url,
                                html_body=difference)
                            logger.info('Sent notification email')

                        old_lines = new_lines

                await asyncio.sleep(10)
```

You'll need to replace the `host`, `sender` and `to` arguments for the mailer component and possibly add the `username` and `password` arguments if your SMTP server requires authentication.

With these changes, you'll get a new HTML formatted email each time the code detects changes in the target web page.

1.2.5 Separating the change detection logic

While the application now works as intended, you're left with two small problems. First off, the target URL and checking frequency are hard coded. That is, they can only be changed by modifying the program code. It is not reasonable to expect non-technical users to modify the code when they want to simply change the target website or the frequency of checks. Second, the change detection logic is hardwired to the notification code. A well designed application should maintain proper separation of concerns. One way to do this is to separate the change detection logic to its own class.

Create a new module named `detector` in the `webnotifier` package. Then, add the change event class to it:

```
import asyncio
import logging

import aiohttp
from async_generator import yield_

from asphalt.core import Component, Event, Signal, context_teardown

logger = logging.getLogger(__name__)


class WebPageChangeEvent(Event):
    def __init__(self, source, topic, old_lines, new_lines):
        super().__init__(source, topic)
        self.old_lines = old_lines
        self.new_lines = new_lines
```

This class defines the type of event that the notifier will emit when the target web page changes. The old and new content are stored in the event instance to allow the event listener to generate the output any way it wants.

Next, add another class in the same module that will do the HTTP requests and change detection:

```
class Detector:
    changed = Signal(WebPageChangeEvent)

    def __init__(self, url, delay):
        self.url = url
        self.delay = delay

    async def run(self):
        with aiohttp.ClientSession() as session:
            last_modified, old_lines = None, None
            while True:
                logger.debug('Fetching contents of %s', self.url)
                headers = {'if-modified-since': last_modified} if last_modified else
↪{}
                async with session.get(self.url, headers=headers) as resp:
                    logger.debug('Response status: %d', resp.status)
                    if resp.status == 200:
                        last_modified = resp.headers['date']
                        new_lines = (await resp.text()).split('\n')
                        if old_lines is not None and old_lines != new_lines:
                            self.changed.dispatch(old_lines, new_lines)

                        old_lines = new_lines

                await asyncio.sleep(self.delay)
```

The constructor arguments allow you to freely specify the parameters for the detection process. The class includes a signal named `changed` that uses the previously created `WebPageChangeEvent` class. The code dispatches such an event when a change in the target web page is detected.

Finally, add the component class which will allow you to integrate this functionality into any Asphalt application:

```
class ChangeDetectorComponent(Component):
    def __init__(self, url, delay=10):
        self.url = url
        self.delay = delay

    @context_teardown
    async def start(self, ctx):
        detector = Detector(self.url, self.delay)
        ctx.add_resource(detector, context_attr='detector')
        task = ctx.loop.create_task(detector.run())
        logging.info('Started web page change detector for url "%s" with a delay of
↪%d seconds',
                     self.url, self.delay)

        # Can be replaced with plain "yield" on Python 3.6+
        await yield_()

        # This part is run when the context is being torn down
        task.cancel()
        await asyncio.gather(task, return_exceptions=True)
        logging.info('Shut down web page change detector')
```

The component's `start()` method starts the detector's `run()` method as a new task, adds the detector object as resource and installs an event listener that will shut down the detector when the context is torn down.

Now that you've moved the change detection code to its own module, `ApplicationComponent` will become somewhat lighter:

```
from async_generator import aclosing


class ApplicationComponent(CLIApplicationComponent):
    async def start(self, ctx):
        self.add_component('detector', ChangeDetectorComponent, url='http://imgur.com
↪')
        self.add_component(
            'mailer', backend='smtp', host='your.smtp.server.here',
            message_defaults={'sender': 'your@email.here', 'to': 'your@email.here'})
        await super().start(ctx)

    async def run(self, ctx):
        diff = HtmlDiff()
        async with aclosing(ctx.detector.changed.stream_events()) as stream:
            async for event in stream:
                difference = diff.make_file(event.old_lines, event.new_lines,␣
↪context=True)
                await ctx.mailer.create_and_deliver(
                    subject='Change detected in %s' % event.source.url, html_
↪body=difference)
                logger.info('Sent notification email')
```

The main application component will now use the detector resource added by `ChangeDetectorComponent`. It adds one event listener which reacts to change events by creating an HTML formatted difference and sending it to the default recipient.

Once the `start()` method here has run to completion, the event loop finally has a chance to run the task created for `Detector.run()`. This will allow the detector to do its work and dispatch those `changed` events that the `page_changed()` listener callback expects.

1.2.6 Setting up the configuration file

Now that your application code is in good shape, you will need to give the user an easy way to configure it. This is where YAML configuration files come in handy. They're clearly structured and are far less intimidating to end users than program code. And you can also have more than one of them, in case you want to run the program with a different configuration.

In your project directory (`tutorial2`), create a file named `config.yaml` with the following contents:

```
---
component:
  type: webnotifier.app:ApplicationComponent
  components:
    detector:
      url: http://imgur.com/
      delay: 15
    mailer:
      host: your.smtp.server.here
      message_defaults:
        sender: your@email.here
        to: your@email.here

logging:
  version: 1
  disable_existing_loggers: false
  formatters:
    default:
      format: '[%(asctime)s %(levelname)s] %(message)s'
  handlers:
    console:
      class: logging.StreamHandler
      formatter: default
  root:
    handlers: [console]
    level: INFO
  loggers:
    webnotifier:
      level: DEBUG
```

The `component` section defines parameters for the root component. Aside from the special `type` key which tells the runner where to find the component class, all the keys in this section are passed to the constructor of `ApplicationComponent` as keyword arguments. Keys under `components` will match the alias of each child component, which is given as the first argument to `add_component()`. Any component parameters given here can now be removed from the `add_component()` call in `ApplicationComponent`'s code.

The logging configuration here sets up two loggers, one for `webnotifier` and its descendants and another (`root`) as a catch-all for everything else. It specifies one handler that just writes all log entries to the standard output. To learn more about what you can do with the logging configuration, consult the Configuration dictionary schema section in the standard library documentation.

You can now run your app with the `asphalt run` command, provided that the project directory is on Python's search path. When your application is properly packaged and installed in `site-packages`, this won't be a problem. But for the purposes of this tutorial, you can temporarily add it to the search path by setting the `PYTHONPATH` environment variable:

```
PYTHONPATH=. asphalt run config.yaml
```

On Windows:

```
set PYTHONPATH=%CD%
asphalt run config.yaml
```

Note: The `if __name__ == '__main__':` block is no longer needed since `asphalt run` is now used as the entry point for the application.

1.2.7 Conclusion

You now know how to take advantage of Asphalt's component system to add structure to your application. You've learned how to build reusable components and how to make the components work together through the use of resources. Last, but not least, you've learned to set up a YAML configuration file for your application and to set up a fine grained logging configuration in it.

You now possess enough knowledge to leverage Asphalt to create practical applications. You are now encouraged to find out what Asphalt component projects exist to aid your application development. Happy coding

CHAPTER 2

User guide

This is the reference documentation. If you're looking to learn Asphalt from scratch, you should take a look at the *Tutorials* first.

2.1 Application architecture

Asphalt applications are centered around the following building blocks:

- components
- contexts
- resources
- signals/events
- the application runner

Components (`Component`) are classes that initialize one or more services, like network servers or database connections and add them to the *context* as *resources*. Components are started by the application runner and usually discarded afterwards.

Contexts (`Context`) are "hubs" through which *resources* are shared between components. Contexts can be chained by setting a parent context for a new context. A context has access to all its parents' resources but parent contexts cannot access the resources of their children.

Resources are any arbitrary objects shared through a context. Every resource is shared on a context using its type (class) and name (chosen by the component). Every combination of type/name is unique in a context.

Signals are the standard way in Asphalt applications to send events to interested parties. Events are dispatched asynchronously without blocking the sender. The signal system was loosely modeled after the signal system in the Qt toolkit.

The *application runner* (`run_application()`) is a function that is used to start an Asphalt application. It configures up the Python logging module, sets up an event loop policy (if configured), creates the root context, starts the root component and then runs the event loop until the application exits. A command line tool (`asphalt`) is provided to better facilitate the running of Asphalt applications. It reads the application configuration from one or more YAML formatted configuration files and calls `run_application()` with the resulting configuration dictionary as keyword arguments. The settings from the configuration file are merged with hard coded defaults so the config file only needs to override settings where necessary.

The following chapters describe in detail how each of these building blocks work.

2.2 Working with components

Components are the basic building blocks of an Asphalt application. They have a narrowly defined set of responsibilities:

1. Take in configuration through the constructor
2. Validate the configuration
3. Add resources to the context (in `start()`)
4. Close/shut down/clean up resources when the context is torn down (by directly adding a callback on the context with `add_teardown_callback()`, or by using `context_teardown()`)

The `start()` method is called either by the parent component or the application runner with a `Context` as its only argument. The component can use the context to add resources for other components and the application business logic to use. It can also request resources provided by other components to provide some complex service that builds on those resources.

The `start()` method of a component is only called once, during application startup. When all components have been started, they are disposed of. If any of the components raises an exception, the application startup process fails and any context teardown callbacks scheduled so far are called before the process is exited.

In order to speed up the startup process and to prevent any deadlocks, components should try to add any resources as soon as possible before requesting any. If two or more components end up waiting on each others' resources, the application will fail to start. Also, if a component needs to perform lengthy operations like connection validation on network clients, it should add all its resources first to avoid the application start timing out.

There is no rule stating that a component cannot add itself to the context as a resource. The reason official Asphalt libraries do not usually do this is that most of them have the option of providing multiple instances of their services, which is obviously not possible when you only add the component itself as a resource.

Hint: It is a good idea to use type hints with typeguard checks (`assert check_argument_types()`) in the component's `__init__` method to ensure that the received configuration values are of the expected type, but this is of course not required.

2.2.1 Container components

A *container component* is component that can contain other Asphalt components. The root component of virtually any nontrivial Asphalt application is a container component. Container components can of course contain other container components and so on.

When the container component starts its child components, each `start()` call is launched in its own task. Therefore all the child components start concurrently and cannot rely on the start order. This is by design. The only way components should be relying on each other is by the sharing of resources in the context.

2.3 Working with contexts and resources

Every Asphalt application has at least one context: the root context. The root context is typically created by the `run_application()` function and passed to the root component. This context will only be closed when the application is shutting down.

Most nontrivial applications will make use of *subcontexts*. A subcontext is a context that has a parent context. A subcontext can make use of its parent's resources, but the parent cannot access the resources of its children. This enables developers to create complex services that work together without risking interfering with each other.

Subcontexts can be roughly divided into two types: long lived and short lived ones. Long lived subcontexts are typically used in container components to isolate its resources from the rest of the application. Short lived subcontexts, on the other hand, usually encompass some *unit of work* (UOW). Examples of such UOWs are:

- handling of a request in a network service
- running a scheduled task
- running a test in a test suite

2.3.1 Resources

The resource system in Asphalt exists for two principal reasons:

- To avoid having to duplicate configuration
- To enable sharing of pooled resources, like database connection pools

Here are a few examples of services that will likely benefit from resource sharing:

- Database connections
- Remote service handles
- Serializers
- Template renderers
- SSL contexts

When you add a resource, you should make sure that the resource is discoverable using any abstract interface or base class that it implements. This is so that consumers of the service don't have to care if you switch the implementation of another. For example, consider a mailer service, provided by asphalt-mailer. The library has an abstract base class for all mailers, `asphalt.mailer.api.Mailer`. To facilitate this loose coupling of services, it adds all its configure mailer services using the `Mailer` interface so that components that just need *some* was to send email don't have to care what implementation was chosen in the configuration.

2.3.2 Adding resources to a context

Resources can be added to a context in two forms: regular resources and resource factories. A regular resource can be any arbitrary object. The same object can be added to the context under several different types, as long as the type/name combination remains unique within the same context.

A resource factory is a callable that takes a `Context` as an argument an returns the value of the resource. There are at least a couple reasons to use resource factories instead of regular resources:

- the resource's lifecycle needs to be bound to the local context (example: database transactions)
- the resource requires access to the local context (example: template renderers)

2.3.3 Getting resources from a context

The `Context` class offers a few ways to look up resources.

The first one, `get_resource()`, looks for a resource or resource factory matching the given type and name. If the resource is found, it returns its value.

The second one, `require_resource()`, works exactly the same way except that it raises `ResourceNotFound` if the resource is not found.

The third method, `request_resource()`, calls `get_resource()` and if the resource is not found, it waits indefinitely for the resource to be added to the context or its parents. When that happens, it calls `get_resource()` again, at which point success is guaranteed. This is usually used only in the components' `start()` methods.

The order of resource lookup is as follows:

1. search for a resource in the local context
2. search for a resource factory in the local context and its parents and, if found, generate the local resource
3. search for a resource in the parent contexts

2.3.4 Handling resource cleanup

Any code that adds resources to a context is also responsible for cleaning them up when the context is closed. This usually involves closing sockets and files and freeing whatever system resources were allocated. This should be done in a *teardown callback*, scheduled using `add_teardown_callback()`. When the context is closed, teardown callbacks are run in the reverse order in which they were added, and always one at a time, unlike with the `Signal` class. This ensures that a resource that is still in use by another resource is never cleaned up prematurely.

For example:

```
from asphalt.core import Component


class FooComponent(Component):
    async def start(ctx):
        service = SomeService()
        await service.start(ctx)
        ctx.add_teardown_callback(service.stop)
        ctx.add_resource(service)
```

There also exists a convenience decorator, `context_teardown()`, which makes use of asynchronous generators:

```
from asphalt.core import Component, context_teardown
from async_generator import yield_


class FooComponent(Component):
    @context_teardown
    async def start(ctx):
        service = SomeService()
        await service.start(ctx)
        ctx.add_resource(service)

        await yield_()  # just "yield" on Python 3.6+

        # This part of the function is run when the context is closing
        service.stop()
```

Sometimes you may want the cleanup to know whether the context was ended because of an unhandled exception. The one use that has come up so far is committing or rolling back a database transaction. This can be achieved by passing the `pass_exception` keyword argument to `add_teardown_callback()`:

```
class FooComponent(Component):
    async def start(ctx):
        def teardown(exception: Optional[BaseException]):
            if exception:
                db.rollback()
            else:
                db.commit()

        db = SomeDatabase()
        await db.start(ctx)
        ctx.add_teardown_callback(teardown, pass_exception=True)
        ctx.add_resource(db)
```

The same can be achieved with `context_teardown()` by storing the yielded value:

```
class FooComponent(Component):
    @context_teardown
    async def start(ctx):
        db = SomeDatabase()
        await db.start(ctx)
        ctx.add_resource(db)

        exception = await yield_()

        if exception:
            db.rollback()
        else:
            db.commit()
```

If any of the teardown callbacks raises an exception, the cleanup process will still continue, but at the end a `TeardownError` will be raised. This exception contains all the raised exceptions in its `exceptions` attribute.

2.4 Working with coroutines and threads

Asphalt was designed as a network oriented framework capable of high concurrency. This means that it can efficiently work with hundreds or even thousands of connections at once. This is achieved by utilizing co-operative multitasking, using an *event loop* provided by the `asyncio` module.

The event loop can only work on one task at a time, so whenever the currently running task needs to wait for something to happen, it will need to explicitly yield control back to the event loop (using `await` and similar statements) to let the event loop run other tasks while this task waits for the result. Once the result is available, the event loop will resume the task.

There is another concurrency mechanism called *threads*. Threads are an implementation of preemptive multitasking, which means that the CPU may run your program at more than one location at once and your code will not have to worry about yielding control to another task. There are some big downsides to using threads, however. First off, threaded code is much more prone to race conditions and programs often need to use locks to share state in a predictable manner. Second, threads don't scale. When you have more threads than CPU cores, the cores need to do context switching, that is, juggle between the threads. With a large number of threads, the overhead from context switching becomes very significant up to the point where the system stops responding altogether.

While Asphalt was designed to avoid the use of threads, they are sometimes necessary. Most third party libraries at the moment don't support the asynchronous concurrency model, and as such, they sometimes need to be used with threads in order to avoid blocking the event loop. Also, file operations cannot, at this time, be executed asynchronously and need to be wrapped in threads. Finally, your application might just need to do some CPU heavy processing that would otherwise block the event loop for long periods of time.

To help with this, Asphalt contains functionality with which you can easily run code in thread pools or call asynchronous code from worker threads.

2.4.1 Examples

Consider a coroutine function that reads the contents of a certain file and then sends them over a network connection. While you might get away with reading the file in the event loop thread, consider what happens if the disk has to spin up from idle state or the file is located on a slow (or temporarily inaccessible) network drive. The whole event loop will then be blocked for who knows how long.

The easiest way is probably to use `call_in_executor()`:

```
from pathlib import Path

async def read_and_send_file(ctx, connection):
    contents = await ctx.call_in_executor(Path('file.txt').read_bytes)
    await connection.send(contents)
```

You can also opt to execute entire blocks with a thread pool executor by using `threadpool()`:

```
async def read_and_send_file(ctx, connection):
    async with ctx.threadpool():
        # Anything inside this block runs in a worker thread!
        contents = Path('file.txt').read_bytes()

    # Don't try to "await" inside the ctx.threadpool() block!
    await connection.send(contents)
```

Alternatively, you can run the whole function in an executor. You will then need to make it a regular function instead of a coroutine function:

```
from asphalt.core import executor

@executor
def read_and_send_file(ctx, connection):
    contents = Path('file.txt').read_bytes()
    ctx.call_async(connection.send, contents)
```

2.4.2 Using alternate executors

By default, all these methods use the default executor of the event loop, which in turn defaults to a `ThreadPoolExecutor` with the default number of workers. Sometimes you may encounter situations where you need to use multiple executors, each earmarked for a particular task or group of tasks so as to prevent other tasks from getting stuck due to the lack of available workers. To this end, the mechanisms described above can be made to target a specific executor, either given directly or acquired as a resource from a context.

Suppose you add an `Executor` resource named `file_ops` to a context:

```
from concurrent.futures import ThreadPoolExecutor, Executor

file_ops = ThreadPoolExecutor(5)  # max 5 worker threads for file operations
ctx.add_resource(file_ops, 'file_ops', types=[Executor])
```

You can then use this executor resource by its name:

```
async def read_and_send_file(ctx, connection):
    contents = await ctx.call_in_executor(Path('file.txt').read_bytes, executor='file_
↪ops')
    await connection.send(contents)
```

Also works with the async context manager:

```
async def read_and_send_file(ctx, connection):
    async with ctx.threadpool('file_ops'):
        contents = Path('file.txt').read_bytes()

    await connection.send(contents)
```

And of course as a decorator too, as long as the context is provided:

```
from asphalt.core import executor

@executor('file_ops')
def read_and_send_file(ctx, connection):
    contents = Path('file.txt').read_bytes()
    ctx.call_async(connection.send, contents)
```

2.5 Working with signals and events

Events are a handy way to make your code react to changes in another part of the application. To dispatch and listen to events, you first need to have one or more `Signal` instances as attributes of some class. Each signal needs to be associated with some `Event` class. Then, when you dispatch a new event by calling `dispatch()`, a new instance of this event class will be constructed and passed to all listener callbacks.

To listen to events dispatched from a signal, you need to have a function or any other callable that accepts a single positional argument. You then pass this callable to `connect()`. That's it!

To disconnect the callback, simply call `disconnect()` with whatever you passed to `connect()` as argument.

Here's how it works:

```
from asphalt.core import Event, Signal


class CustomEvent(Event):
    def __init__(source, topic, extra_argument):
        super().__init__(source, topic)
        self.extra_argument = extra_argument


class MyEventSource:
    somesignal = Signal(Event)
    customsignal = Signal(CustomEvent)
```

(continues on next page)

(continued from previous page)

```
def plain_listener(event):
    print('received event: %s' % event)


async def coro_listener(event):
    print('coroutine listeners are fine too: %s' % event)


async def some_handler():
    source = MyEventSource()
    source.somesignal.connect(plain_listener)
    source.customsignal.connect(coro_listener)

    # Dispatches an Event instance
    source.somesignal.dispatch()

    # Dispatches a CustomEvent instance (the extra argument is passed to its␣
↪constructor)
    source.customsignal.dispatch('extra argument here')
```

2.5.1 Exception handling

Any exceptions raised by the listener callbacks are logged to the `asphalt.core.event` logger. Additionally, the future returned by `dispatch()` resolves to `True` if no exceptions were raised during the processing of listeners. This was meant as a convenience for use with tests where you can just do `assert await thing.some_signal.dispatch('foo')`.

2.5.2 Waiting for a single event

To wait for the next event dispatched from a given signal, you can use the `wait_event()` method:

```
async def print_next_event(source):
    event = await source.somesignal.wait_event()
    print(event)
```

You can even wait for the next event dispatched from any of several signals using the `wait_event()` function:

```
from asphalt.core import wait_event


async def print_next_event(source1, source2, source3):
    event = await wait_event(source1.some_signal, source2.another_signal, source3.
↪some_signal)
    print(event)
```

As a convenience, you can provide a filter callback that will cause the call to only return when the callback returns `True`:

```
async def print_next_matching_event(source1, source2, source3):
    event = await wait_event(source1.some_signal, source2.another_signal, source3.
↪some_signal,
```

(continues on next page)

(continued from previous page)

```
                                lambda event: event.myrandomproperty == 'foo')
    print(event)
```

2.5.3 Receiving events iteratively

With `stream_events()`, you can even asynchronously iterate over events dispatched from a signal:

```
from async_generator import aclosing


async def listen_to_events(source):
    async with aclosing(source.somesignal.stream_events()) as stream:
        async for event in stream:
            print(event)
```

Using `stream_events()`, you can stream events from multiple signals:

```
from asphalt.core import stream_events


async def listen_to_events(source1, source2, source3):
    stream = stream_events(source1.some_signal, source2.another_signal, source3.some_
↪signal)
    async with aclosing(stream):
        async for event in stream:
            print(event)
```

The filtering capability of `wait_event()` works here too:

```
async def listen_to_events(source1, source2, source3):
    stream = stream_events(source1.some_signal, source2.another_signal, source3.some_
↪signal,
                           lambda event: event.randomproperty == 'foo')
    async with aclosing(stream):
        async for event in stream:
            print(event)
```

2.6 Testing Asphalt components

Testing Asphalt components and component hierarchies is a relatively simple procedure:

1. Create an instance of your `Component`
2. Create a `Context` instance
3. Run the component's `start()` method with the context as the argument
4. Run the tests
5. Close the context to release any resources

With Asphalt projects, it is recommended to use the py.test testing framework because it is already being used with Asphalt core and it provides easy testing of asynchronous code (via the pytest-asyncio plugin).

2.6.1 Example

Let's build a test suite for the *Echo Tutorial*.

The client and server components could be tested separately, but to make things easier, we'll test them against each other.

Create a `tests` directory at the root of the project directory and create a module named `test_client_server` there (the `test_` prefix is important):

```
import asyncio

import pytest
from asphalt.core import Context

from echo.client import ClientComponent
from echo.server import ServerComponent


@pytest.fixture
def event_loop():
    # Required on pytest-asyncio v0.4.0 and newer since the event_loop fixture␣
↪provided by the
    # plugin no longer sets the global event loop
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    yield loop
    loop.close()


@pytest.fixture
def context(event_loop):
    with Context() as ctx:
        yield ctx


@pytest.fixture
def server_component(event_loop, context):
    component = ServerComponent()
    event_loop.run_until_complete(component.start(context))


def test_client(event_loop, server_component, context, capsys):
    client = ClientComponent('Hello!')
    event_loop.run_until_complete(client.start(context))
    exc = pytest.raises(SystemExit, event_loop.run_forever)
    assert exc.value.code == 0

    # Grab the captured output of sys.stdout and sys.stderr from the capsys fixture
    out, err = capsys.readouterr()
    assert out == 'Message from client: Hello!\nServer responded: Hello!\n'
```

The test module above contains one test function (`test_client`) and three fixtures:

- `event_loop`: provides an asyncio event loop and closes it after the test
- `context` provides the root context and runs teardown callbacks after the test
- `server_component`: creates and starts the server component

The client component is not provided as a fixture because, as always with `CLIApplicationComponent`, starting it would run the logic we want to test, so we defer that to the actual test code.

In the test function (`test_client`), the client component is instantiated and started. Since the component's `start()` function only kicks off the task that runs the client's business logic (the `run()` method), we have to wait until the task is complete by running the event loop (using `run_forever()`) until `run()` finishes and its callback code attempts to terminate the application. For that purpose, we catch the resulting `SystemExit` exception and verify that the application indeed completed successfully, as indicated by the return code of 0.

Finally, we check that the server and the client printed the messages they were supposed to. When the server receives a line from the client, it prints a message to standard output using `print()`. Likewise, when the client gets a response from the server, it too prints out its own message. By using pytest's built-in `capsys` fixture, we can capture the output and verify it against the expected lines.

To run the test suite, make sure you're in the project directory and then do:

```
pytest tests
```

For more elaborate examples, please see the test suites of various Asphalt subprojects.

2.7 Configuration and deployment

As your application grows more complex, you may find that you need to have different settings for your development environment and your production environment. You may even have multiple deployments that all need their own custom configuration.

For this purpose, Asphalt provides a command line interface that will read a YAML formatted configuration file and run the application it describes.

2.7.1 Running the Asphalt launcher

Running the launcher is very straightfoward:

```
asphalt run yourconfig.yaml [your-overrides.yml...]
```

Or alternatively:

python -m asphalt run yourconfig.yaml [your-overrides.yml…]

What this will do is:

1. read all the given configuration files, starting from `yourconfig.yaml`
2. **merge the configuration files' contents into a single configuration dictionary using** `merge_config()`
3. **call `run_application()` using the configuration dictionary as keyword** arguments

2.7.2 Writing a configuration file

A production-ready configuration file should contain at least the following options:

- `component`: a dictionary containing the class name and keyword arguments for its constructor
- `logging`: a dictionary to be passed to `logging.config.dictConfig()`

Suppose you had the following component class as your root component:

```
class MyRootComponent(ContainerComponent):
    def __init__(self, components, data_directory: str):
        super().__init__(components)
        self.data_directory = data_directory

    async def start(ctx):
        self.add_component('mailer', backend='smtp')
        self.add_component('sqlalchemy')
        await super().start(ctx)
```

You could then write a configuration file like this:

```
---
max_threads: 20
component:
  type: myproject:MyRootComponent
  data_directory: /some/file/somewhere
  components:
    mailer:
      host: smtp.mycompany.com
      ssl: true
    sqlalchemy:
      url: postgresql:///mydatabase

logging:
  version: 1
  disable_existing_loggers: false
  handlers:
    console:
      class: logging.StreamHandler
      formatter: generic
  formatters:
    generic:
        format: "%(asctime)s:%(levelname)s:%(name)s:%(message)s"
  root:
    handlers: [console]
    level: INFO
```

In the above configuration you have three top level configuration keys: `max_threads`, `component` and `logging`, all of which are directly passed to `run_application()` as keyword arguments.

The `component` section defines the type of the root component using the specially processed `type` option. You can either specify a setuptools entry point name (from the `asphalt.components` namespace) or a text reference like `module:class` (see `resolve_reference()` for details). The rest of the keys in this section are passed directly to the constructor of the `MyRootComponent` class.

The `components` section within `component` is processed in a similar fashion. Each subsection here is a component type alias and its keys and values are the constructor arguments to the relevant component class. The per-component configuration values are merged with those provided in the `start()` method of `MyRootComponent`. See the next section for a more elaborate explanation.

With `max_threads: 20`, the maximum number of threads in the event loop's default thread pool executor is set to 20.

The `logging` configuration tree here sets up a root logger that prints all log entries of at least `INFO` level to the console. You may want to set up more granular logging in your own configuration file. See the Python standard library documentation for details.

2.7.3 Using data from environment variables and files

Many deployment environments (Kubernetes, Docker Swarm, Heroku, etc.) require applications to input configuration values and/or secrets using environment variables or external files. To support this, Asphalt extends the YAML parser with three custom tags:

- `!Env`: substitute with the value of an environment variable
- `!TextFile` substitute with the contents of a (UTF-8 encoded) text file (as `str`)
- `!BinaryFile` substitute with the contents of a file (as `bytes`)

For example:

```yaml
---
component:
  type: myproject:MyRootComponent
  param_from_environment: !Env MY_ENV_VAR
  files:
    - !TextFile /path/to/file.txt
    - !BinaryFile /path/to/file.bin
```

If a file path contains spaces, you can just quote it:

```yaml
---
component:
  type: myproject:MyRootComponent
  param_from_text_file: !TextFile "/path with spaces/to/file.txt"
```

Note: This does **not** allow you to include other YAML documents as part of the configuration, except as text/binary blobs. See the next section if this is what you want.

New in version 4.5.0.

2.7.4 Configuration overlays

Component configuration can be specified on several levels:

- Hard-coded arguments to `add_component()`
- First configuration file argument to `asphalt run`
- Second configuration file argument to `asphalt run`
- ...

Any options you specify on each level override or augment any options given on previous levels. To minimize the effort required to build a working configuration file for your application, it is suggested that you pass as many of the options directly in the component initialization code and leave only deployment specific options like API keys, access credentials and such to the configuration file.

With the configuration presented in the earlier paragraphs, the `mailer` component's constructor gets passed three keyword arguments:

- `backend='smtp'`
- `host='smtp.mycompany.com'`
- `ssl=True`

The first one is provided in the root component code while the other two options come from the YAML file. You could also override the mailer backend in the configuration file if you wanted. The same effect can be achieved programmatically by supplying the override configuration to the container component via its `components` constructor argument. This is very useful when writing tests against your application. For example, you might want to use the `mock` mailer in your test suite configuration to test that the application correctly sends out emails (and to prevent them from actually being sent to recipients!).

There is another neat trick that lets you easily modify a specific key in the configuration. By using dotted notation in a configuration key, you can target a specific key arbitrarily deep in the configuration structure. For example, to override the logging level for the root logger in the configuration above, you could use an override configuration such as:

```yaml
---
logging.root.level: DEBUG
```

The keys don't need to be on the top level either, so the following has the same effect:

```yaml
---
logging:
    root.level: DEBUG
```

2.7.5 Defining multiple services

New in version 4.1.0.

Sometimes it may be more convenient to use a single configuration file for launching your application with different configurations or entry points. To this end, the runner supports the notion of "service definitions" in the configuration file. This is done by replacing the `component` dictionary with a `services` dictionary at the top level of the configuration file and either setting the `ASPHALT_SERVICE` environment variable or by passing the `--service` (or `-s`) option when launching the runner. This approach provides the additional advantage of allowing the use of YAML references, like so:

```yaml
---
services:
  server:
    max_threads: 30
    component:
      type: myproject.server.ServerComponent
      components:
        wamp: &wamp
          host: wamp.example.org
          port: 8000
          tls: true
          auth_id: serveruser
          auth_secret: serverpass
        mailer:
          backend: smtp
  client:
    component:
      type: myproject.client.ClientComponent
      components:
        wamp:
          <<: *wamp
          auth_id: clientuser
          auth_secret: clientpass
```

Each section under `services` is like its own distinct top level configuration. Additionally, the keys under each service are merged with any top level configuration, so you can, for example, define a logging configuration there.

Now, to run the `server` service, do:

```
asphalt run -s server config.yaml
```

The `client` service is run in the same fashion:

```
asphalt run -s client config.yaml
```

You can also define a service with a special name, `default`, which is used in case multiple services are present and no service has been explicitly selected.

Note: The `-s/--service` command line switch overrides the `ASPHALT_SERVICE` environment variable.

2.7.6 Performance tuning

Asphalt's core code and many third part components employ a number of potentially expensive validation steps in its code. The performance hit of these checks is not a concern in development and testing, but in a production environment you will probably want to maximize the performance.

To do this, you will want to disable Python's debugging mode by either setting the environment variable `PYTHONOPTIMIZE` to `1` or (if applicable) running Python with the `-O` switch. This has the effect of completely eliminating all `assert` statements and blocks starting with `if __debug__:` from the compiled bytecode.

When you want maximum performance, you'll also want to use the fastest available event loop implementation. This can be done by specifying the `event_loop_policy` option in the configuration file or by using the `-l` or `--loop` switch. The core library has built-in support for the uvloop event loop implementation, which should provide a nice performance boost over the standard library implementation.

CHAPTER 3

Version history

This library adheres to Semantic Versioning.

4.5.0 (2019-03-26)

- Added new custom YAML tags (`!Env`, `!BinaryFile` and `!TextFile`)

4.4.4 (2018-05-08)

- Changed the `async_timeout` dependency to allow the 3.x and newer releases

4.4.3 (2018-02-05)

- Fixed exception in `stream_events()` cleanup code introduced in the previous release

4.4.2 (2018-02-02)

- Fixed memory leak when `stream_events()` is called but the returned generator is never used

4.4.1 (2018-01-21)

- Fixed incompatibility with Python 3.5.2

4.4.0 (2017-11-25)

- Removed the requirement for async generators to yield at least once when wrapped with `@context_teardown`
- Removed aiogevent support since it has been removed from PyPI

4.3.0 (2017-11-05)

- The runner now calls `logging.shutdown()` after the event loop has been closed
- Added the `Context.get_resources()` method
- Made `stream_events()` connect to the signal when called instead of the first iteration of the async generator

4.2.0 (2017-08-24)

- Allowed selecting the service to run with `asphalt run` using an environment variable (`ASPHALT_SERVICE`)

4.1.0 (2017-08-18)

- Added support for the Tokio event loop
- Added a feature to the runner that lets one define multiple services in a configuration file and select which one to run
- Increased the runner default start timeout to 10 seconds

4.0.0 (2017-06-04)

- **BACKWARD INCOMPATIBLE** When a teardown callback raises an exception during `Context.close()`, a `TeardownException` is raised at the end instead of the error being logged
- Renamed the `asphalt.core.command` module to `asphalt.core.cli`
- Fixed the inability to override the component type from external configuration (contributed by Devin Fee)

3.0.2 (2017-05-05)

- Fixed `CLIApplicationComponent` running prematurely (during the application start phase) and skipping the proper shutdown sequence
- Fixed return code from `CLIApplicationComponent` being ignored

3.0.1 (2017-04-30)

- Fixed `run_application()` not working on Windows due to `NotImplementedError` when adding the `SIGTERM` signal handler

3.0.0 (2017-04-10)

- **BACKWARD INCOMPATIBLE** Upped the minimum Python version to 3.5.2 from 3.5.0
- **BACKWARD INCOMPATIBLE** Renamed the `asphalt.core.util` module to `asphalt.core.utils`
- The `asphalt.core.event` module was overhauled:
  - **BACKWARD INCOMPATIBLE** Removed the `monotime` attribute from the `Event` class
  - **BACKWARD INCOMPATIBLE** Dropped the `return_future` argument from `Signal.dispatch()` and `Signal.dispatch_event()` – they now always return an awaitable that resolves to a boolean, indicating whether all callbacks were successful or not
  - **BACKWARD INCOMPATIBLE** Made the `max_queue_size` argument in `Signal.stream_events` and `stream_events()` into a keyword-only argument
  - **BACKWARD INCOMPATIBLE** `Signal.dispatch_event()` was renamed to `Signal.dispatch_raw()`
  - Added the `filter` argument to `Signal.stream_events()` and `stream_events()` which can restrict the events that are yielded by them
  - Added the `time` constructor argument to the `Event` class
- The `asphalt.core.context` module was overhauled:
  - "lazy resources" are now called "resource factories"
  - `Context.get_resources()` now returns a set of `ResourceContainer` (instead of a list)
  - **BACKWARD INCOMPATIBLE** The `default_timeout` parameter was removed from the `Context` constructor
  - **BACKWARD INCOMPATIBLE** The `timeout` parameter of `Context.request_resource()` was removed

  - **BACKWARD INCOMPATIBLE** The `alias` parameter of `Context.request_resource()` was renamed to `name`
  - **BACKWARD INCOMPATIBLE** Removed the `Context.finished` signal in favor of the new `add_teardown_callback()` method which has different semantics (callbacks are called in LIFO order and awaited for one at a time)
  - **BACKWARD INCOMPATIBLE** Removed the ability to remove resources from a `Context`
  - Added several new methods to the `Context` class: `close()`, `get_resource()`, `require_resource()`
  - **BACKWARD INCOMPATIBLE** `Context.publish_resource()` was renamed to `Context.add_resource()`
  - **BACKWARD INCOMPATIBLE** `Context.publish_lazy_resource()` was renamed to `Context.add_resource_factory()`
  - **BACKWARD INCOMPATIBLE** The `Context.get_resources()` method was removed until it can be replaced with a better thought out API
  - **BACKWARD INCOMPATIBLE** The `Resource` class was removed from the public API
  - Three new methods were added to the `Context` class to bridge `asyncio_extras` and `Executor` resources: `call_async()`, `call_in_executor()` and `threadpool()`
  - Added a new decorator, `@executor` to help run code in specific `Executor` resources
- The application runner (`asphalt.core.runner`) got some changes too:
  - **BACKWARD INCOMPATIBLE** The runner no longer cancels all active tasks on exit
  - **BACKWARD INCOMPATIBLE** There is now a (configurable, defaults to 5 seconds) timeout for waiting for the root component to start
  - Asynchronous generators are now closed after the context has been closed (on Python 3.6+)
  - The SIGTERM signal now cleanly shuts down the application
- Switched from `asyncio_extras` to `async_generator` as the async generator compatibility library
- Made the current event loop accessible (from any thread) as the `loop` property from any `asphalt.core.context.Context` instance to make it easier to schedule execution of async code from worker threads
- The `asphalt.core.utils.merge_config()` function now accepts `None` as either argument (or both)

2.1.1 (2017-02-01)

- Fixed memory leak which prevented objects containing Signals from being garbage collected
- Log a message on startup that indicates whether optimizations (`-O` or `PYTHONOPTIMIZE`) are enabled

2.1.0 (2016-09-26)

- Added the possibility to specify more than one configuration file on the command line
- Added the possibility to use the command line interface via `python -m asphalt ...`
- Added the `CLIApplicationComponent` class to facilitate the creation of Asphalt based command line tools
- Root component construction is now done after installing any alternate event loop policy provider
- Switched YAML library from PyYAML to ruamel.yaml
- Fixed a corner case where in `wait_event()` the future's result would be set twice, causing an exception in the listener

- Fixed coroutine-based lazy resource returning a CoroWrapper instead of a Future when asyncio's debug mode has been enabled
- Fixed a bug where a lazy resource would not be created separately for a context if a parent context contained an instance of the same resource

2.0.0 (2016-05-09)

- **BACKWARD INCOMPATIBLE** Dropped Python 3.4 support in order to make the code fully rely on the new `async`/`await`, `async for` and `async with` language additions
- **BACKWARD INCOMPATIBLE** De-emphasized the ability to implicitly run code in worker threads. As such, Asphalt components are no longer required to transparently work outside of the event loop thread. Instead, use `asyncio_extras.threads.call_async()` to call asynchronous code from worker threads if absolutely necessary. As a direct consequence of this policy shift, the `asphalt.core.concurrency` module was dropped in favor of the `asyncio_extras` library.
- **BACKWARD INCOMPATIBLE** The event system was completely rewritten:
  - instead of inheriting from `EventSource`, event source classes now simply assign `Signal` instances to attributes and use `object.signalname.connect()` to listen to events
  - all event listeners are now called independently of each other and coroutine listeners are run concurrently
  - added the ability to stream events
  - added the ability to wait for a single event to be dispatched
- **BACKWARD INCOMPATIBLE** Removed the `asphalt.command` module from the public API
- **BACKWARD INCOMPATIBLE** Removed the `asphalt quickstart` command
- **BACKWARD INCOMPATIBLE** Removed the `asphalt.core.connectors` module
- **BACKWARD INCOMPATIBLE** Removed the `optional` argument of `Context.request_resource()`
- **BACKWARD INCOMPATIBLE** Removed the `asphalt.core.runners` entry point namespace
- **BACKWARD INCOMPATIBLE** `Component.start()` is now required to be a coroutine method
- **BACKWARD INCOMPATIBLE** Removed regular context manager support from the `Context` class (asynchronous context manager support still remains)
- **BACKWARD INCOMPATIBLE** The `Context.publish_resource()`, `Context.publish_lazy_resource()` and `Context.remove_resource()` methods are no longer coroutine methods
- **BACKWARD INCOMPATIBLE** Restricted resource names to alphanumeric characters and underscores
- Added the possibility to specify a custom event loop policy
- Added support for uvloop
- Added support for aiogevent
- Added the ability to use coroutine functions as lazy resource creators (though that just makes them return a `Future` instead)
- Added the ability to get a list of all the resources in a Context
- Changed the `asphalt.core.util.resolve_reference()` function to return invalid reference strings as-is
- Switched from argparse to click for the command line interface
- All of Asphalt core's public API is now importable directly from `asphalt.core`

1.2.0 (2016-01-02)

- Moved the `@asynchronous` and `@blocking` decorators to the `asphalt.core.concurrency` package along with related code (they're still importable from `asphalt.core.util` until v2.0)
- Added typeguard checks to fail early if arguments of wrong types are passed to functions

1.1.0 (2015-11-19)

- Decorated `ContainerComponent.start` with `@asynchronous` so that it can be called by a blocking subclass implementation
- Added the `stop_event_loop` function to enable blocking callables to shut down Asphalt's event loop

1.0.0 (2015-10-18)

- Initial release

CHAPTER 4

Acknowledgements

Many thanks to following people for the time spent helping with Asphalt's development:

- Alice Bevan-McGregor (brainstorming and documentation QA)
- Guillaume "Cman" Brun (brainstorming)
- Darin Gordon (brainstorming and documentation QA)
- Antti Haapala (brainstorming)
- Olli Paloheimo (Asphalt logo design).
- Cody Scott (tutorials QA)
- API reference