The Asphalt Framework (core)

Release 2.1.1

Apr 09, 2017

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.

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.

Prerequisites

Asphalt requires Python 3.5.0 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).

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
```

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
```

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).

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.

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
```

```
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
```

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.

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.

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).

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 start(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.

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 start(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 = '\n'.join(unified_diff(old_lines, new_lines))
                            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.

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)
                            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 `ssl`, `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.

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_finisher

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 `change` 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_finisher
    async def start(self, ctx):
        detector = Detector(self.url, self.delay)
        ctx.publish_resource(detector, context_attr='detector')
        task = asyncio.get_event_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 finished
        task.cancel()
        logging.info('Shut down web page change detector')
```

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

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

```
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 for event in ctx.detector.changed.stream_events():
            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 published 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.

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
  loggers:
    root:
      handlers: [console]
      level: INFO
    webnotifier:
      handlers: [console]
      level: DEBUG
      propagate: false
```

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.

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.

Application architecture

Asphalt applications are built by assembling a hierarchy of *components*. Each component typically provides some specific functionality for the application, like a network server or client, a database connection or a myriad of other things. A component's lifecycle is usually very short: it's instantiated and its `start()` method is run and the component is then discarded. A common exception to this are command line tools, where the root component's `start()` call typically lasts for the entire run time of the tool.

Components work together through a shared `Context`. Every application has at least a top level context which is passed to the root component's `start()` method. A context is essentially a container for *resources* and a namespace for arbitrary data attributes. Resources can be objects of any type like data or services.

Contexts can have subcontexts. How and if subcontexts are used depends on the components using them. For example, a component serving network requests may want to create a subcontext for each request it handles to store request specific information and other state. While the subcontext will have its own independent state, it also has full access the resources of its parent context.

An Asphalt application is normally started by calling `run_application()` with the root component as the argument. This function takes care of logging and and starting the root component in the event loop. The application will then run until Ctrl+C is pressed, the process is terminated from outside or the application code stops the event loop.

The runner is further extended by the `asphalt` command line tool which reads the application configuration from a YAML formatted configuration file, instantiates the root component and calls `run_application()`. The settings from the configuration file are merged with hard coded defaults so the config file only needs to override settings where necessary.

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. Publish resources (in `start()`)
4. Close/shut down/cleanup resources when the context is finished (by adding a callback on the `finished` signal of the context, or by using `context_finisher()`)

In the `start()` method, the component receives a `Context` as its only argument. The component can use the context to publish 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 the context is finished.

In order to speed up the startup process and to prevent any deadlocks, components should try to publish 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 due to timeout errors. Also, if a component needs to perform lengthy operations like connection validation on network clients, it should publish all its resources first to avoid said timeouts.

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.

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 publishing and requesting of resources in their shared context.

Context hierarchies

As mentioned previously, every application has at least one context. Component code and application business logic can create new contexts at any time, and a new context can be linked to a parent context to take advantage of its resources. Such *subcontexts* have access to all the resources of the parent context, but parent contexts cannot access resources from their subcontexts. Sometimes it may also be beneficial to create completely isolated contexts to ensure consistent behavior when some reusable code is plugged in an application.

A common use case for creating subcontexts is when a network server handles an incoming request. Such servers typically want to create a separate subcontext for each request, usually using specialized subclass of `Context`.

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 publish 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 publishes all mailers as Mailers.

Lazy resources

Resources can also be published *lazily*. That means they're created *on demand*, that is, either when their context attribute is accessed or when the resource is being requested for the first time. Unlike with normal resources, the resource values are not inherited by subcontexts, but every time the resource is requested in a new context, a new value is created specifically for that context.

There are at least a couple plausible reasons for publishing resources this way:

- The resource needs access to the resources or data specific to the local context (example: template renderers)
- The life cycle of the resource needs to be tied to the life cycle of the context (example: database transactions)

Lazy resources are published using `publish_lazy_resource()`. Instead of passing a static value to it, you give it a callable that takes the local context object (whatever that may be) as the argument and returns the created resource object. The creator callable will only be called at most once per context.

The creator callable can be a coroutine function or return an awaitable, in which case the coroutine or other awaitable is resolved before returning the resource object to the caller. This approach has the unfortunate limitation that the awaitable cannot be automatically resolved on attribute access so something like `await ctx.resourcename` is required when such resources are accessed through their context attributes.

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, the asyncio_extras library was created as a byproduct of Asphalt. It provides several conveniences you can use to easily use threads when the need arises.

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 `open_async()`:

```
from asyncio_extras import open_async

async def read_and_send_file(connection):
    async with open_async('file.txt', 'rb') as f:
        contents = await f.read()

    await connection.send(contents)
```

The following snippet achieves the same goal:

```
from asyncio_extras import threadpool

async def read_and_send_file(connection):
    async with threadpool():
        with open('file.txt', 'rb') as f:
            contents = f.read()

    await connection.send(contents)
```

As does the next one:

```
from asyncio_extras import call_in_executor

async def read_and_send_file(connection):
    f = await call_in_executor(open, 'file.txt', 'rb')
    with f:
        contents = await call_in_executor(f.read)

    await connection.send(contents)
```

Alternatively, you can run the whole function in the thread pool. You will need to make it a regular function instead of a coroutine function and you must explicitly pass in the event loop object:

```
from asyncio_extras import threadpool, call_async

@threadpool
def read_and_send_file(connection, loop):
    with open('file.txt', 'rb') as f:
        contents = f.read()

    call_async(loop, connection.send, contents)
```

Using alternate thread pools

In more advanced applications, you may find it useful to set up specialized thread pools for certain tasks in order to avoid the default thread pool from being overburdened:

```
from concurrent.futures import ThreadPoolExecutor

from asyncio_extras import threadpool

file_ops = ThreadPoolExecutor(5)  # max 5 threads for file operations


async def read_and_send_file(connection):
    async with threadpool(file_ops):
        with open('file.txt', 'rb') as f:
            contents = f.read()

    await connection.send(contents)
```

All the thread related utilities in asyncio_extras have a way to specify the executor to use. Refer to its documentation for the specifics.

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)


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')
```

Exception handling

By default, all exceptions raised by listener callbacks are just sent to the logger (`asphalt.core.event`). If the dispatcher needs to know about any exceptions raised by listeners, it can call `dispatch()` with `return_future=True`. This will cause a `Future` to be returned and, when awaited, will raise a `EventDispatchError` if any listener raised an exception. This exception will contain every exception that was raised, along with the information regarding which callback raised which exception.

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)
```

Receiving events iteratively

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

```
async def listen_to_events(source):
    async for event in source.somesignal.stream_events():
        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):
    async for event in stream_events(source1.some_signal, source2.another_signal,
                                     source3.some_signal):
        print(event)
```

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. Dispatch the `finished` event on 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).

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):
    ctx = Context()
    yield ctx
    event_loop.run_until_complete(ctx.finished.dispatch(None, return_future=True))


@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 finish 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.

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.

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

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:

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

```
      url: postgresql:///mydatabase
max_threads: 20
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: `component`, `max_threads` 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.

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:

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

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

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

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.

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