Dependency Injection in Nameko

Matt Bennett ~ Senior Engineer ~ onefinestay

Dependency Injection

Design pattern whereby an object's dependencies are injected or passed by reference into it, rather than constructed by the object.

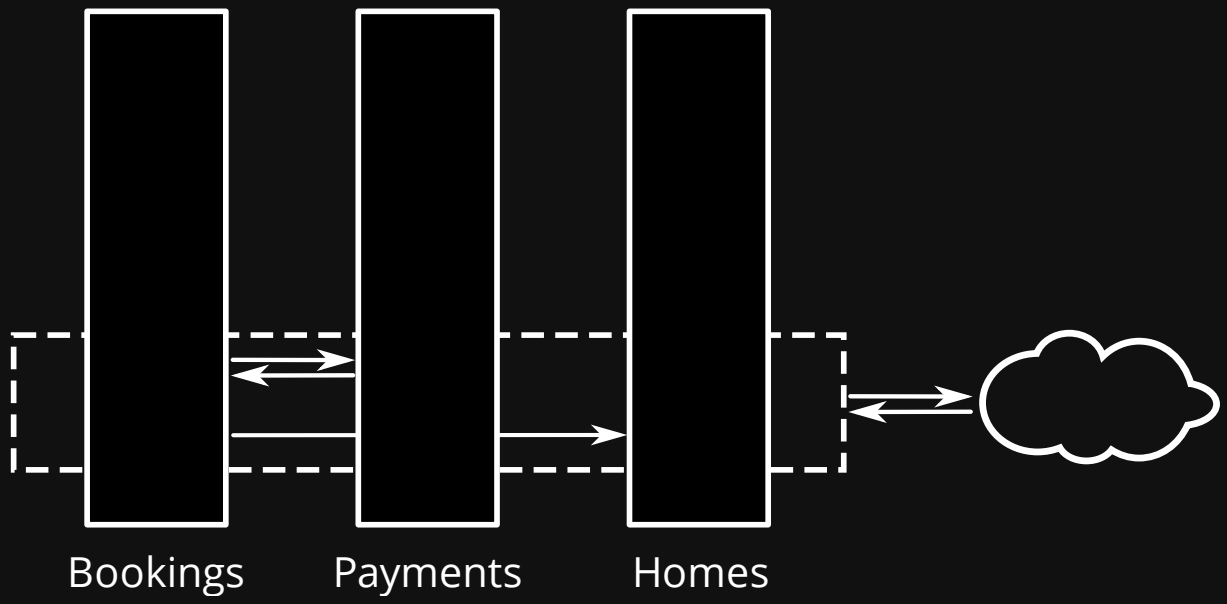
```
class Widget(object):
    def __init__(self):
        self.session = get_db_session()
```

```
class Widget(object):
    def __init__(self, session):
        self.session = session
```

Pretty simple

Nameko

Python framework for building service-oriented software



- Open-source software
- Active development on GitHub

Nameko out of the box

Everything you need to do:

- AMQP RPC
- AMQP Pub-Sub

Definitions

- Entrypoints
  - To interact with a service
- Dependencies
  - For the service to interact with external things

Example Service

```
from nameko.rpc import rpc
from .dependencies import database_connection


class HelloService(object):

    database = database_connection(DB_URI)

    @rpc
    def hello(self, name):
        self.database.add(name)
        return "Hello, {}".format(name)
```

Running HelloService

```
from nameko.containers import ServiceContainer

config = {'AMQP_URI': 'amqp://guest:guest@localhost:5672/'}
container = ServiceContainer(HelloService, ..., config)
container.start()
...
```

Worker Lifecycle

1. Entrypoint "fires"
2. Worker created
3. Dependencies injected
4. Execute method
5. Discard worker

```
worker_instance = HelloService()
```

```
worker_instance.database = HelloService.database.inject()
```

```
result = worker_instance.hello(name)
```

```
del worker_instance
```

Why do it this way?

- Simple service logic
- Complex application

```
from nameko.rpc import rpc
from .dependencies import database_connection


class HelloService(object):

    database = database_connection(DB_URI)

    @rpc
    def hello(self, name):
        self.database.add(name)
        return "Hello, {}".format(name)
```

Separation of concerns

Unit Testing

Testing a service in isolation

```
from mock import call
from nameko.testing.services import worker_factory

hello_svc = worker_factory(HelloService)

assert hello_svc.hello("Matt") == "Hello, Matt"
assert hello_svc.database.add.call_args_list == [call("Matt")]
```

```
from nameko.testing.services import worker_factory

database = set()
hello_svc = worker_factory(HelloService, database=database)

assert hello_svc.hello("Matt") == "Hello, Matt"
assert database == {"Matt"}
```

Integration Testing

Testing a service with something else

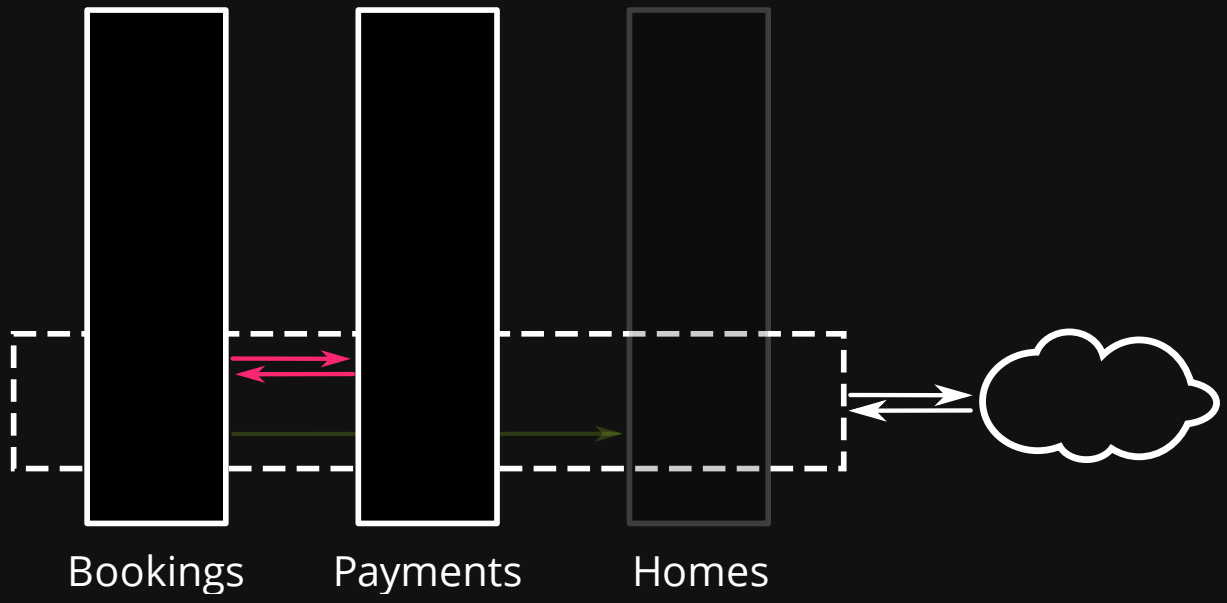
- Host services the "normal" way
- Restrict scope of interaction through dependencies

Replace Injections

```
from nameko.testing.services import replace_injections

container = ServiceContainer(BookingService, ...)
publisher_mock = replace_injections(container, "homes_publisher")

container.start()
...
```



Worker Lifecycle

All dependencies participate

- worker setup
- worker result
- worker teardown

Nice Examples

- Log Aggregator
- Sentry Reporter
- Statsd Monitor

Summary

- Dependency Injection
  - Simple idea
  - Separation of concerns
- Nameko
  - Encourages explicit dependencies
  - Provides nice testing tools
  - And other goodies...

Nameko Workshop -- Tomorrow 10:20, MADE room

Thanks!

Questions?

https://github.com/onefinestay/nameko

P.S. onefinestay are hiring...