The 101 guide to deploying Django

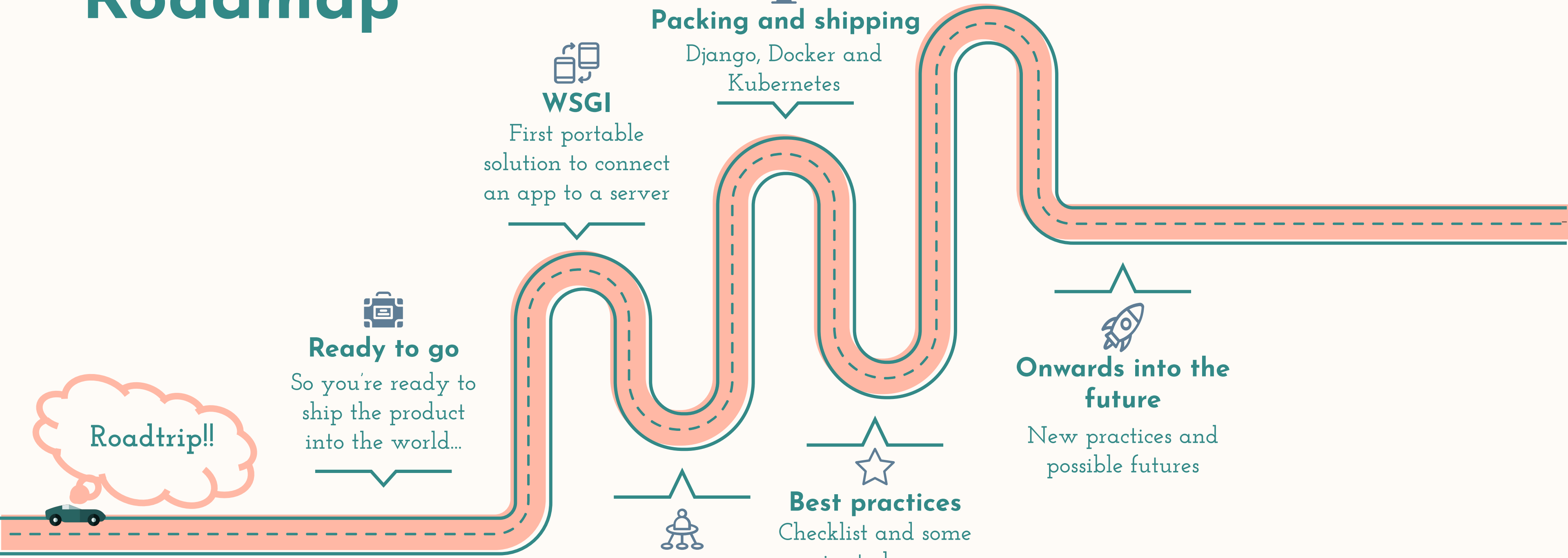
Iulia Avram

Django Day Copenhagen 2020

whoami

- developer
- curious as a cat

Roadmap



Ready to go
So you're ready to ship the product into the world...

WSGI
First portable solution to connect an app to a server

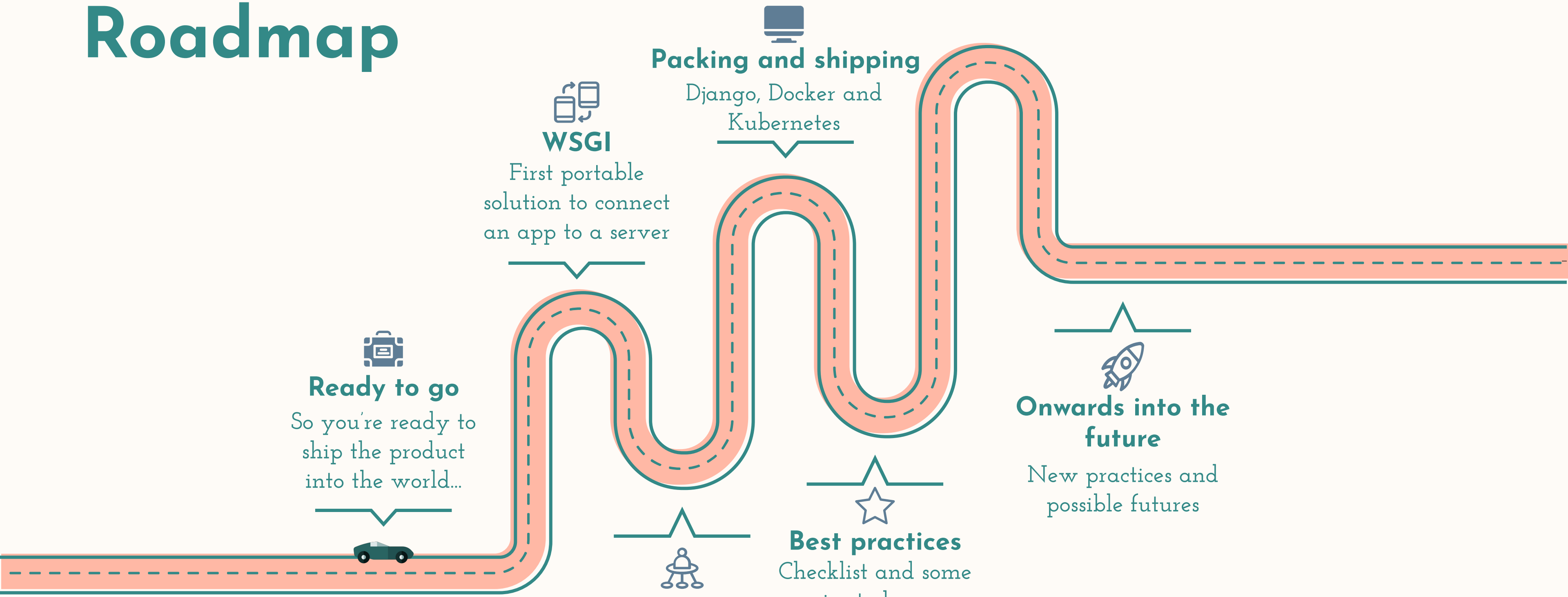
ASGI
The power of async

Packing and shipping
Django, Docker and Kubernetes

Best practices
Checklist and some nice to haves

Onwards into the future
New practices and possible futures

Roadmap



Ready to go
So you're ready to ship the product into the world...

WSGI
First portable solution to connect an app to a server

ASGI
The power of async

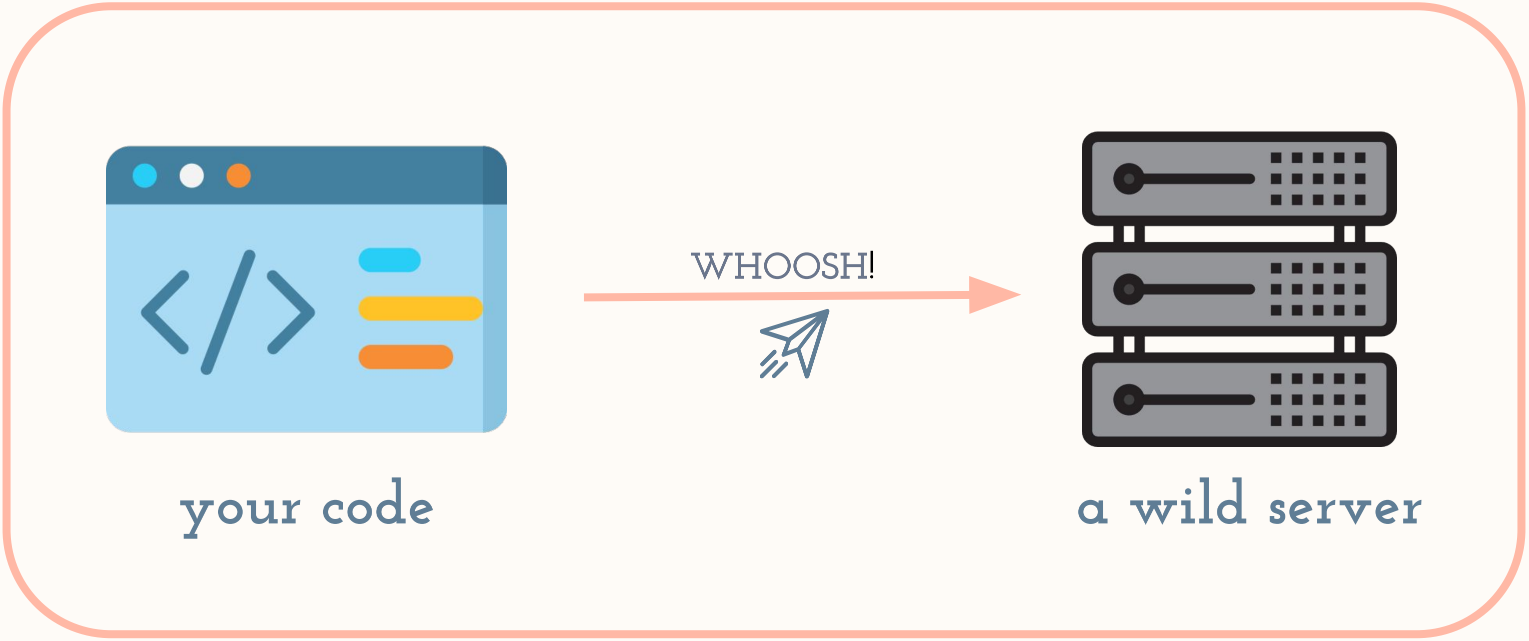
Packing and shipping
Django, Docker and Kubernetes

Best practices
Checklist and some nice to haves

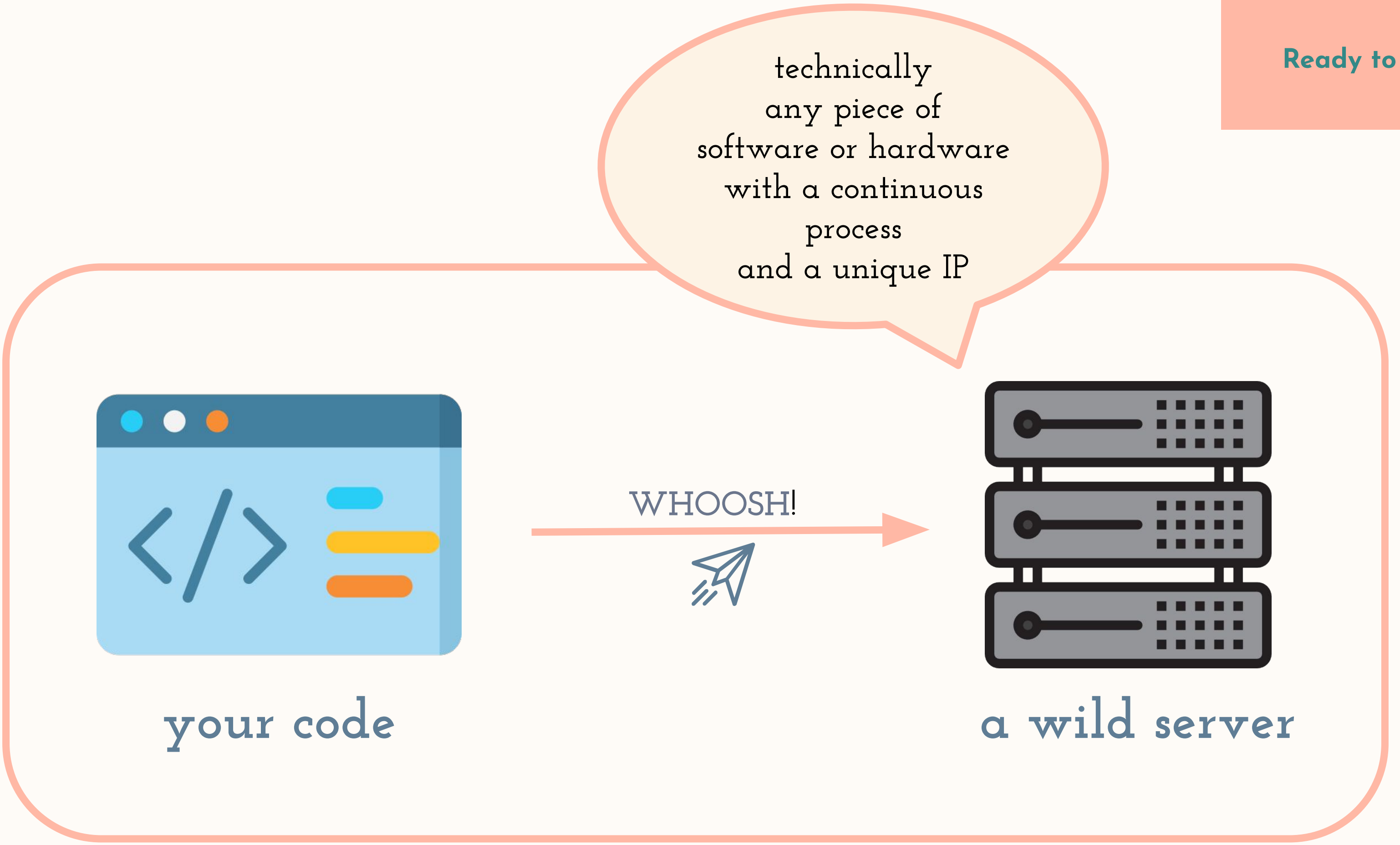
Onwards into the future
New practices and possible futures

What happens when you deploy an application?

Ready to go



Ready to go


technically any piece of software or hardware with a continuous process and a unique IP
WHOOSH!
your code
a wild server

Ready to go

uWSGI

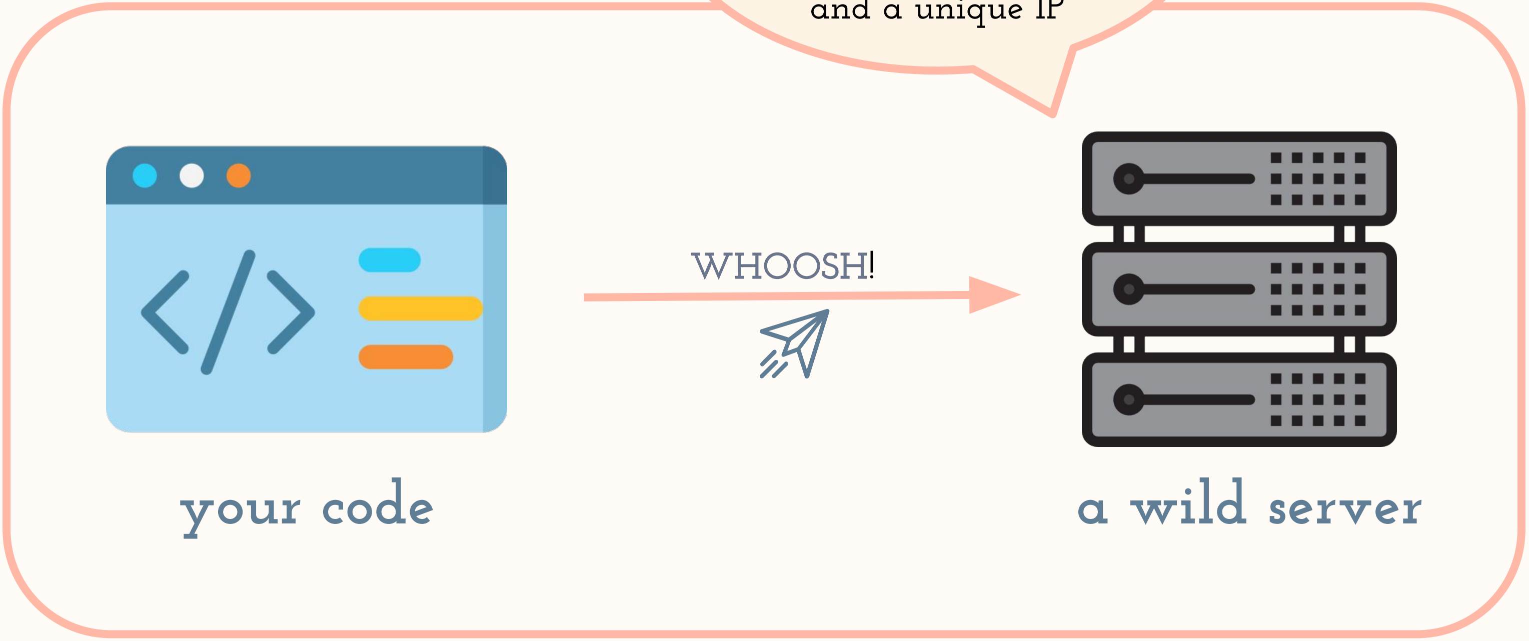
N

Apache

uvicorn

Hypercorn

technically
any piece of
software or hardware
with a continuous
process
and a unique IP



How does the code get to the wild server?

1. You can install it directly 🤭

Ready to go

1. You can install it directly 🤭

2. Use a container for easy replication (such as Docker) 📦

Ready to go

1. You can install it directly 🤭

2. Use a container for easy replication (such as Docker) 📦

3. Go serverless ☁️

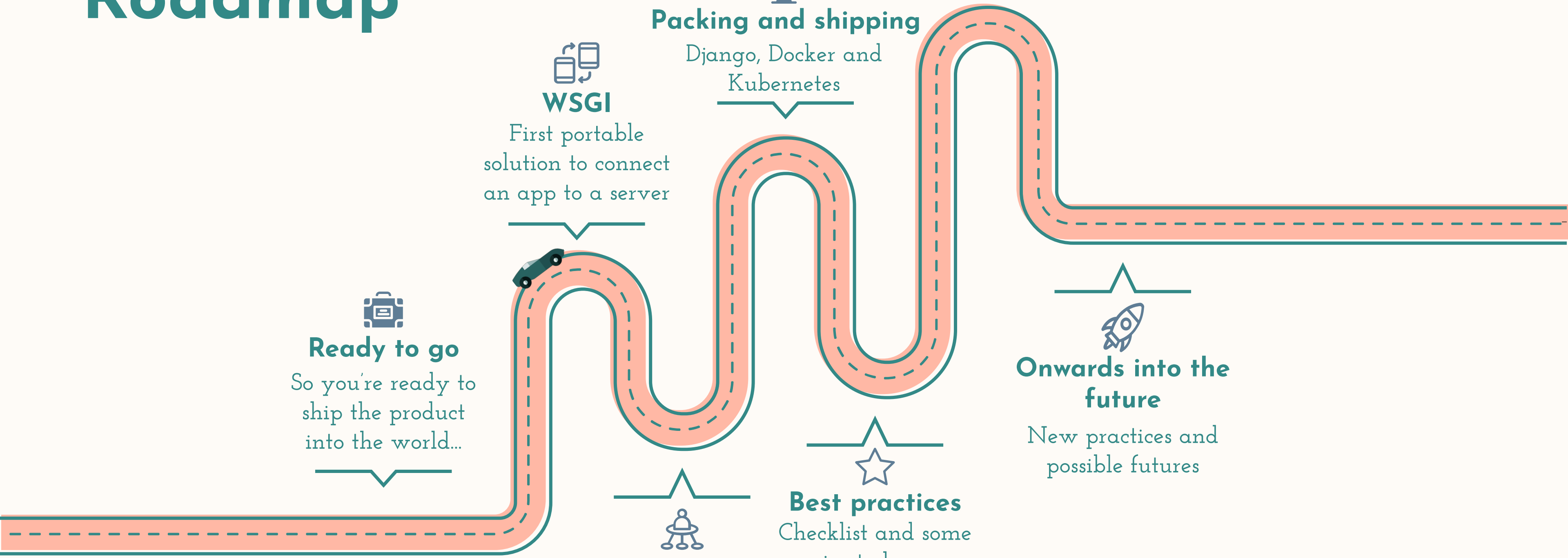
But before that, we need a server for the server...

The Django documentation mentions two main methods of deploying

:

- WSGI $\longrightarrow$ only supports synchronous code
- ASGI $\longrightarrow$ asynchronous-friendly

Roadmap



Ready to go

So you're ready to ship the product into the world…

WSGI

First portable solution to connect an app to a server

ASGI

The power of async

Packing and shipping

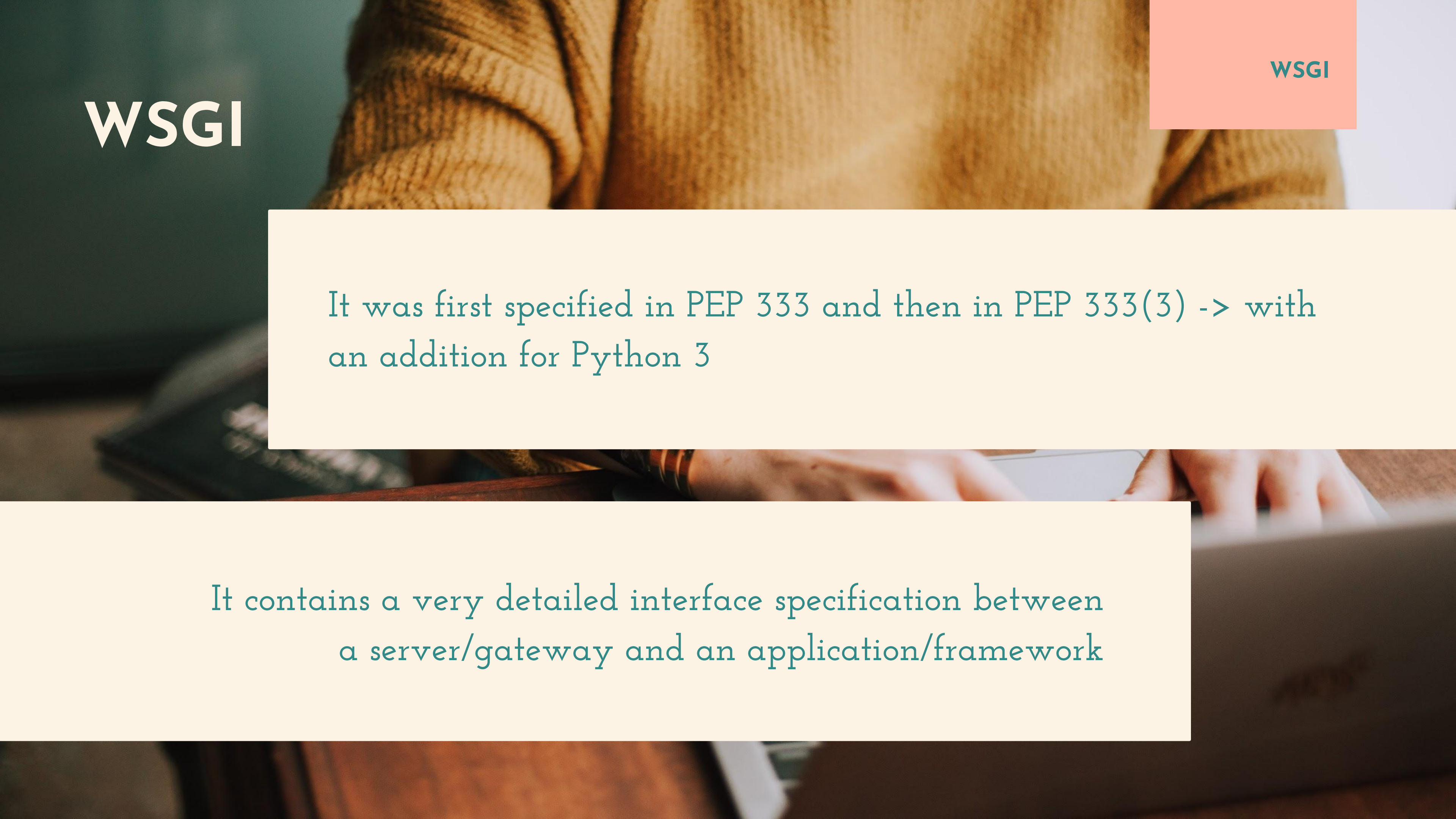
Django, Docker and Kubernetes

Best practices

Checklist and some nice to haves

Onwards into the future

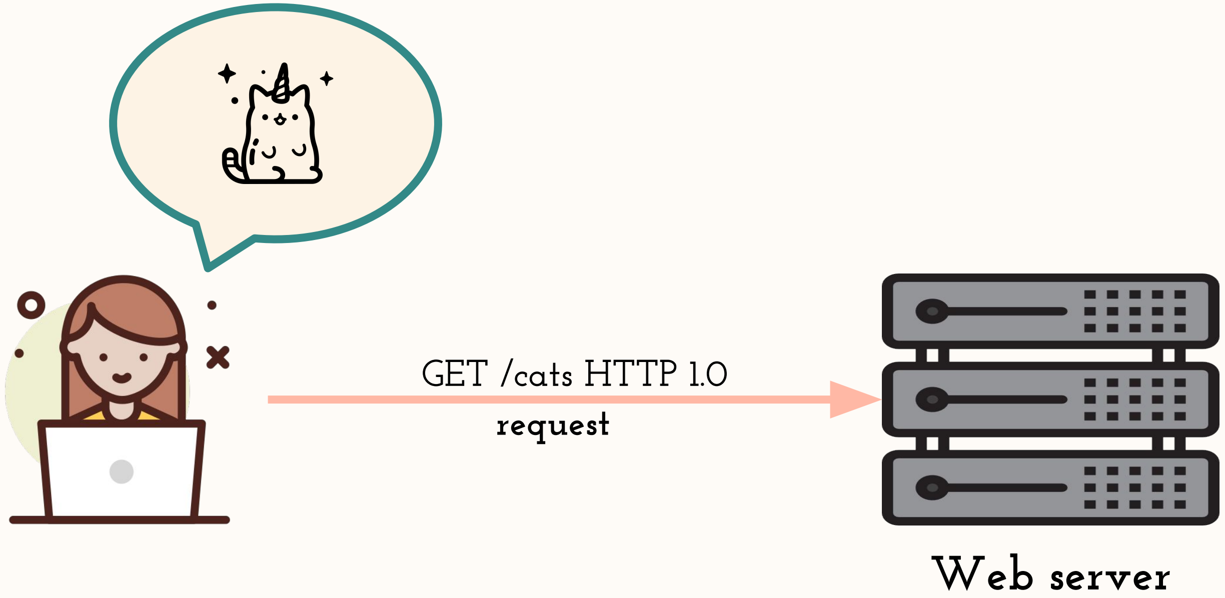
New practices and possible futures



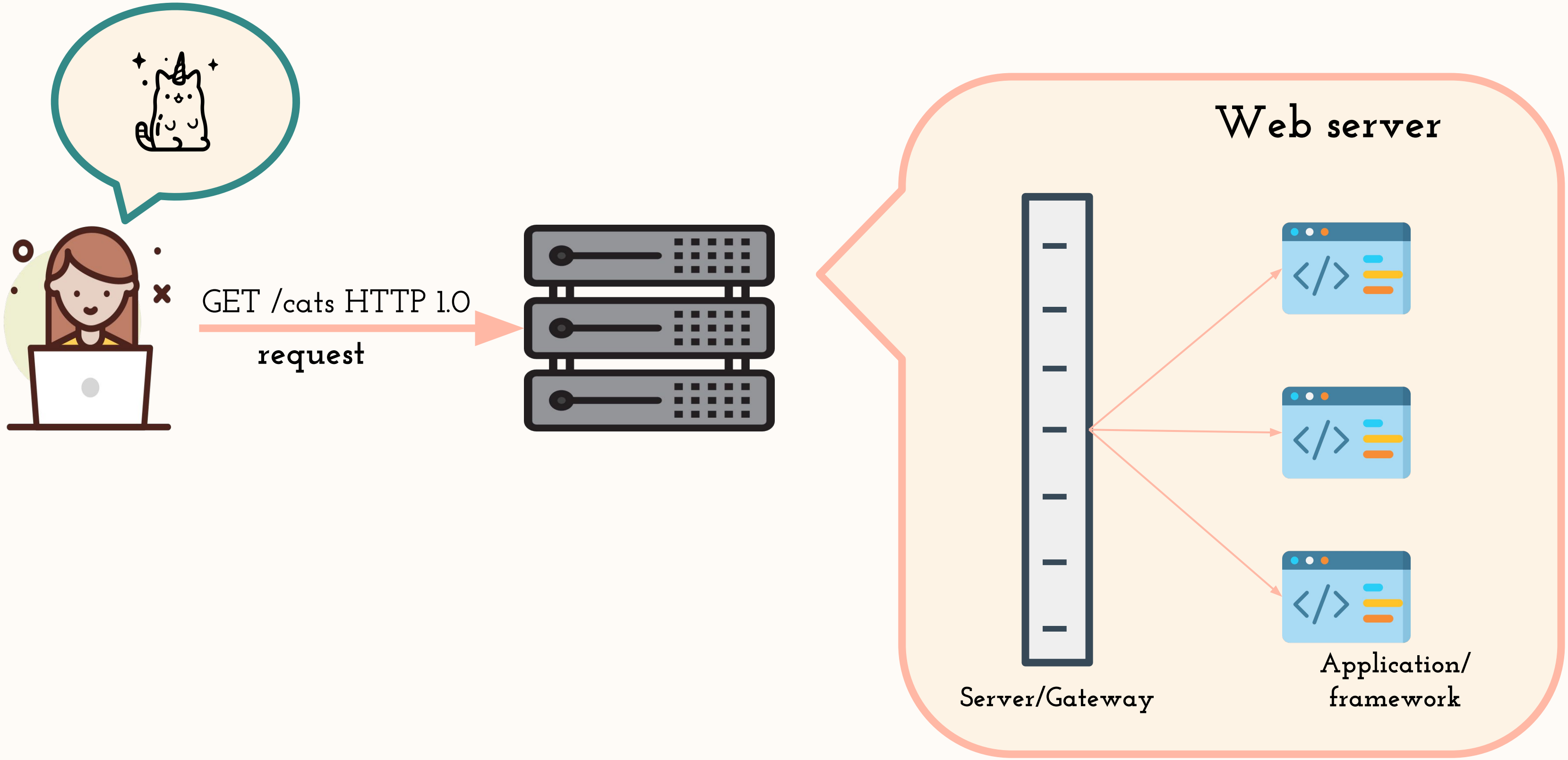
WSGI

It was first specified in PEP 333 and then in PEP 333(3) -> with an addition for Python 3

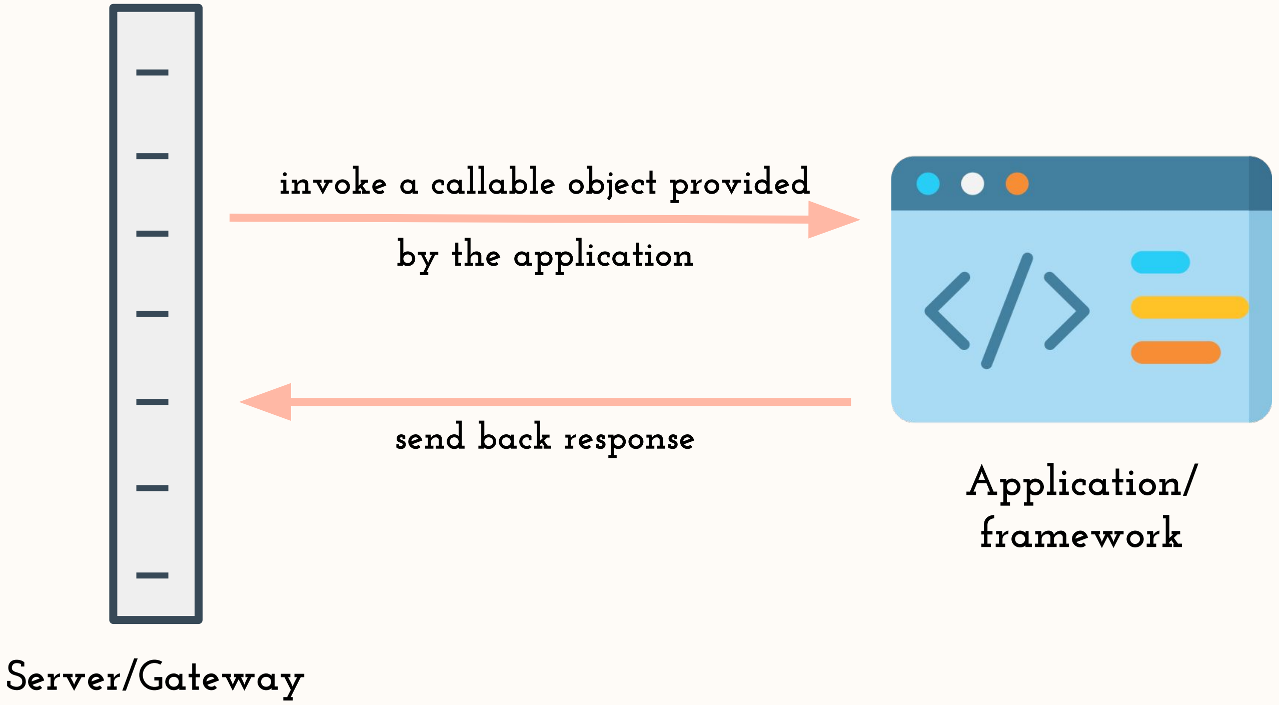
It contains a very detailed interface specification between a server/gateway and an application/framework


GET /cats HTTP 1.0
request
Web server

WSGI


GET /cats HTTP 1.0
request
Web server
Server/Gateway
Application/
framework

WSGI


invoke a callable object provided
by the application
send back response
Server/Gateway
Application/
framework

WSGI

```
import os

from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'copenhagen.settings')

application = get_wsgi_application()
```

WSGI

```python
import os

from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'copenhagen.settings')

application = get_wsgi_application()
```

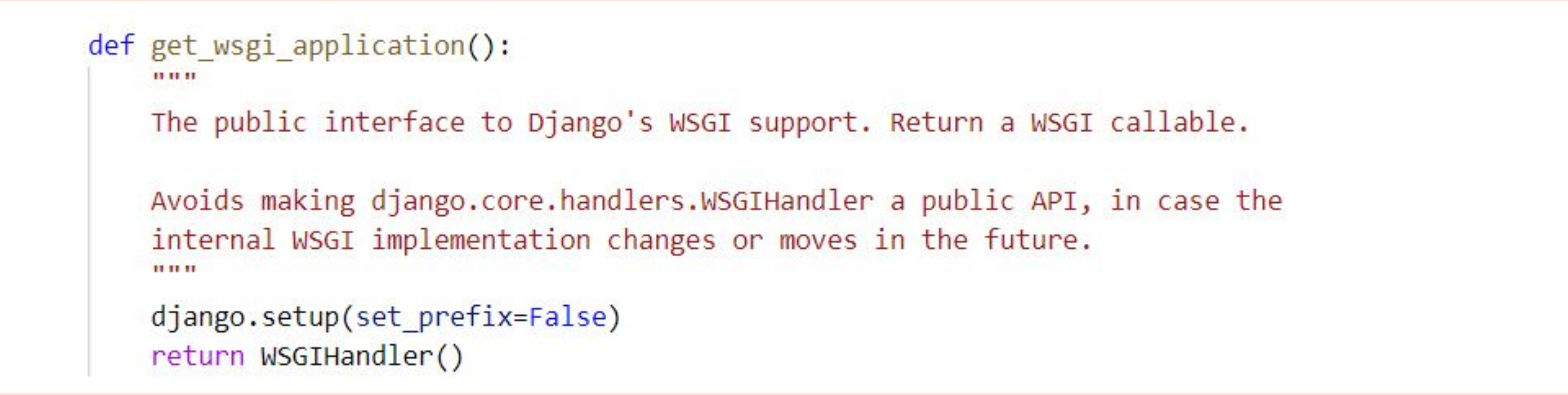
WSGI

```python
import os

from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'copenhagen.settings')

application = get_wsgi_application()
```

WSGI

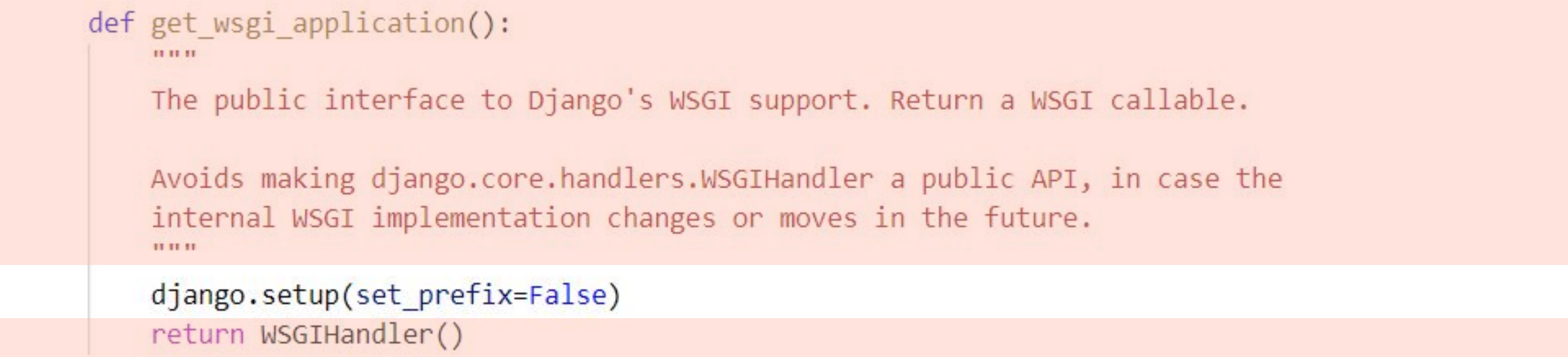
```
import os

from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'copenhagen.settings')

application = get_wsgi_application(
```

WSGI

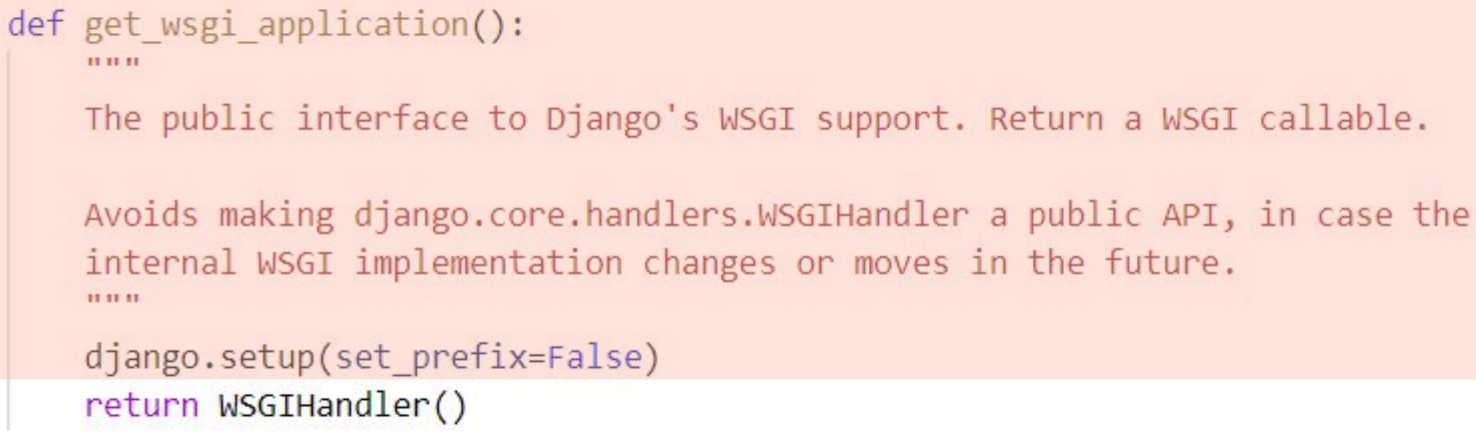


```
def get_wsgi_application():
    """
    The public interface to Django's WSGI support. Return a WSGI callable.

    Avoids making django.core.handlers.WSGIHandler a public API, in case the
    internal WSGI implementation changes or moves in the future.
    """
    django.setup(set_prefix=False)
    return WSGIHandler()
```

WSGI

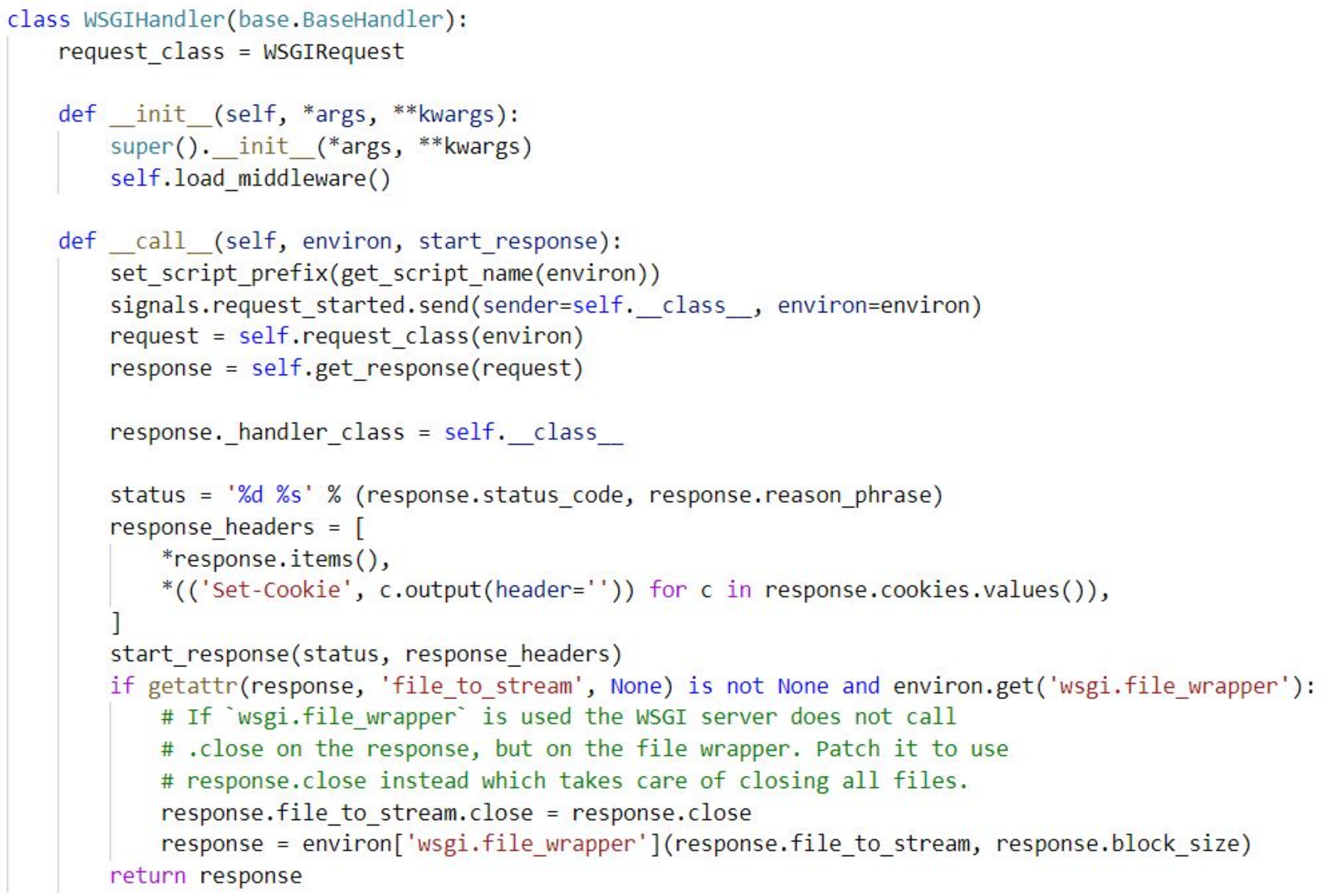


```python
def get_wsgi_application():
    """
    The public interface to Django's WSGI support. Return a WSGI callable.

    Avoids making django.core.handlers.WSGIHandler a public API, in case the
    internal WSGI implementation changes or moves in the future.
    """
    django.setup(set_prefix=False)
    return WSGIHandler()
```



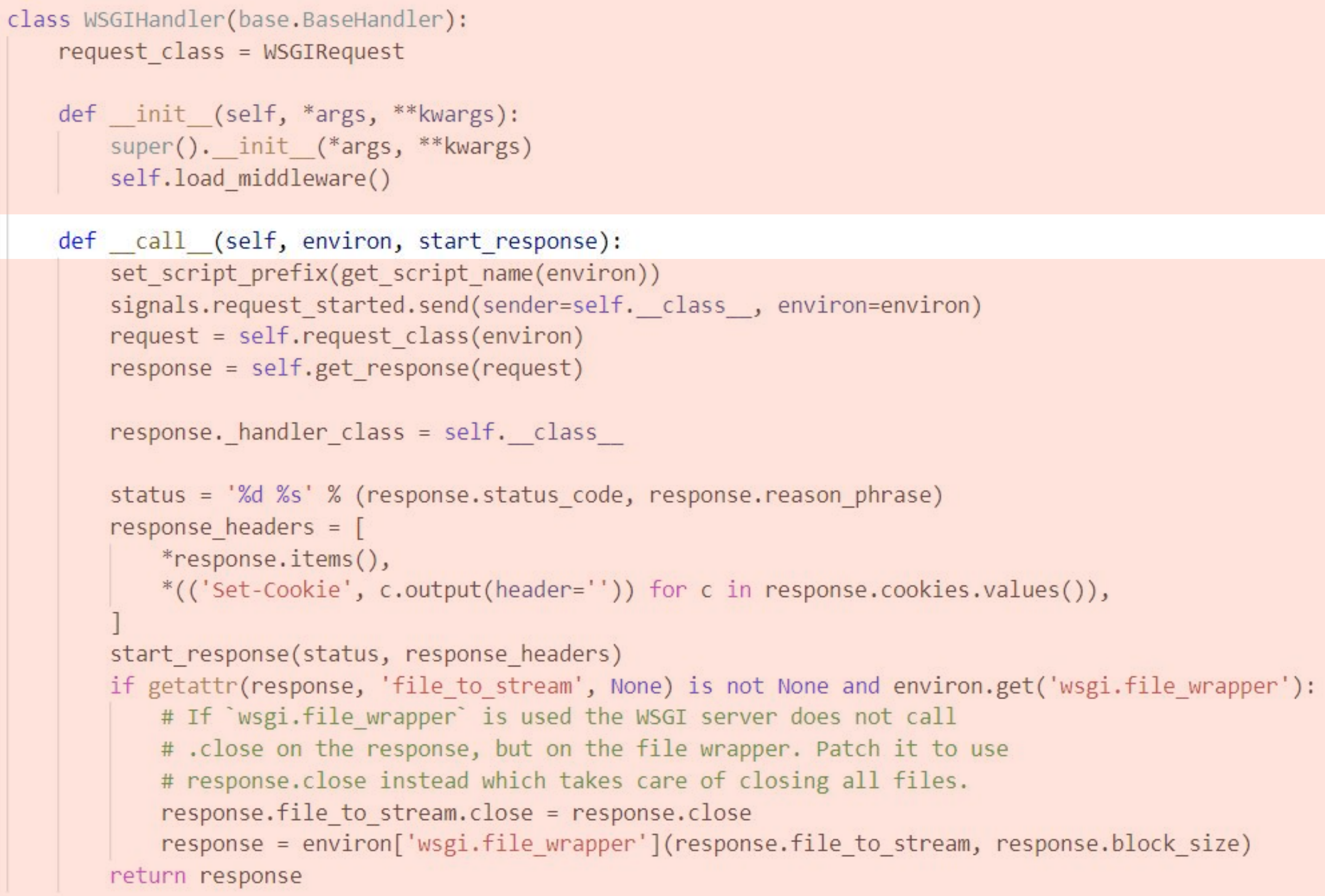
```
def get_wsgi_application():
    """
    The public interface to Django's WSGI support. Return a WSGI callable.

    Avoids making django.core.handlers.WSGIHandler a public API, in case the
    internal WSGI implementation changes or moves in the future.
    """
    django.setup(set_prefix=False)
    return WSGIHandler()
```

WSGI



```python
class WSGIHandler(base.BaseHandler):
    request_class = WSGIRequest

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.load_middleware()

    def __call__(self, environ, start_response):
        set_script_prefix(get_script_name(environ))
        signals.request_started.send(sender=self.__class__, environ=environ)
        request = self.request_class(environ)
        response = self.get_response(request)

        response._handler_class = self.__class__

        status = '%d %s' % (response.status_code, response.reason_phrase)
        response_headers = [
            *response.items(),
            *(('Set-Cookie', c.output(header='')) for c in response.cookies.values()),
        ]
        start_response(status, response_headers)
        if getattr(response, 'file_to_stream', None) is not None and environ.get('wsgi.file_wrapper'):
            # If `wsgi.file_wrapper` is used the WSGI server does not call
            # .close on the response, but on the file wrapper. Patch it to use
            # response.close instead which takes care of closing all files.
            response.file_to_stream.close = response.close
            response = environ['wsgi.file_wrapper'](response.file_to_stream, response.block_size)
        return response
```

WSGI



```python
class WSGIHandler(base.BaseHandler):
    request_class = WSGIRequest

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.load_middleware()

    def __call__(self, environ, start_response):
        set_script_prefix(get_script_name(environ))
        signals.request_started.send(sender=self.__class__, environ=environ)
        request = self.request_class(environ)
        response = self.get_response(request)

        response._handler_class = self.__class__

        status = '%d %s' % (response.status_code, response.reason_phrase)
        response_headers = [
            *response.items(),
            *(('Set-Cookie', c.output(header='')) for c in response.cookies.values()),
        ]
        start_response(status, response_headers)
        if getattr(response, 'file_to_stream', None) is not None and environ.get('wsgi.file_wrapper'):
            # If `wsgi.file_wrapper` is used the WSGI server does not call
            # .close on the response, but on the file wrapper. Patch it to use
            # response.close instead which takes care of closing all files.
            response.file_to_stream.close = response.close
            response = environ['wsgi.file_wrapper'](response.file_to_stream, response.block_size)
        return response
```

WSGI ARGUMENTS

1

environ

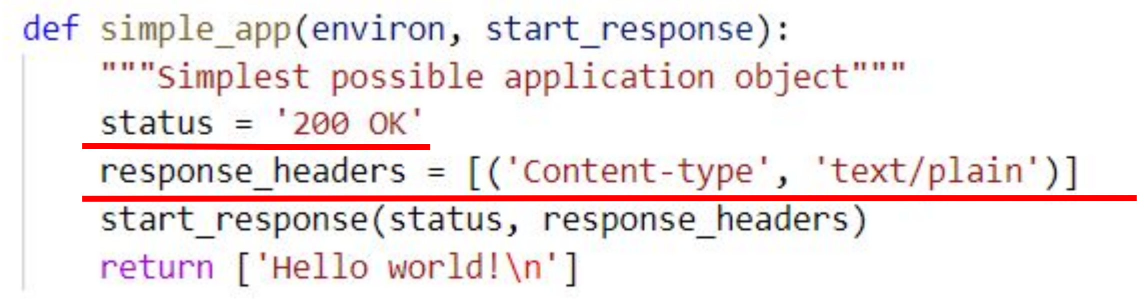
dictionary object containing CGI-style environment variables

2

start_response

callable accepting 2 positional arguments and one optional - **status**: string, **response_headers**: list of tuples containing (header_name, value) and **exc_info**: used with errors

WSGI example



```
def simple_app(environ, start_response):
    """Simplest possible application object"""
    status = '200 OK'
    response_headers = [('Content-type', 'text/plain')]
    start_response(status, response_headers)
    return ['Hello world!\n']
```

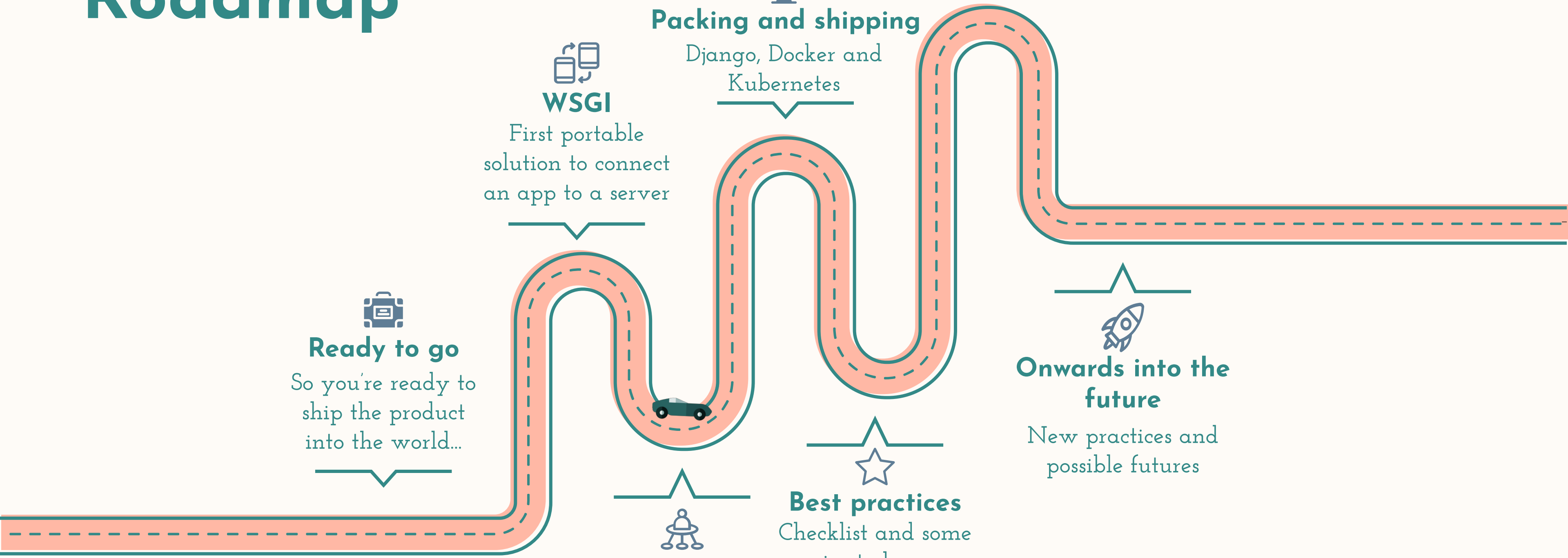
Source: https://www.python.org/dev/peps/pep-0333/

Limitations of WSGI

- it's synchronous
    - no websockets
    - no await/async

- only works with the HTTP protocol

Roadmap



Ready to go
So you're ready to ship the product into the world...

WSGI
First portable solution to connect an app to a server

ASGI
The power of async

Packing and shipping
Django, Docker and Kubernetes

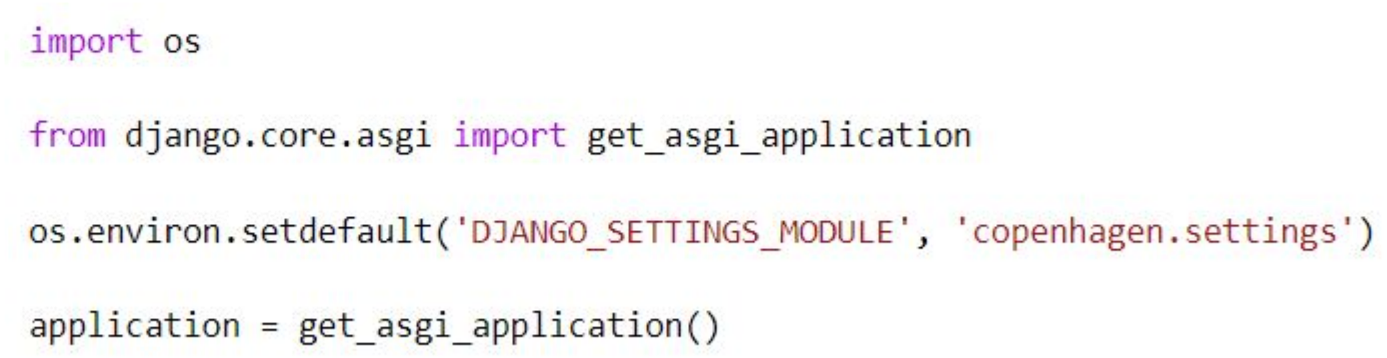
Best practices
Checklist and some nice to haves

Onwards into the future
New practices and possible futures

ASGI

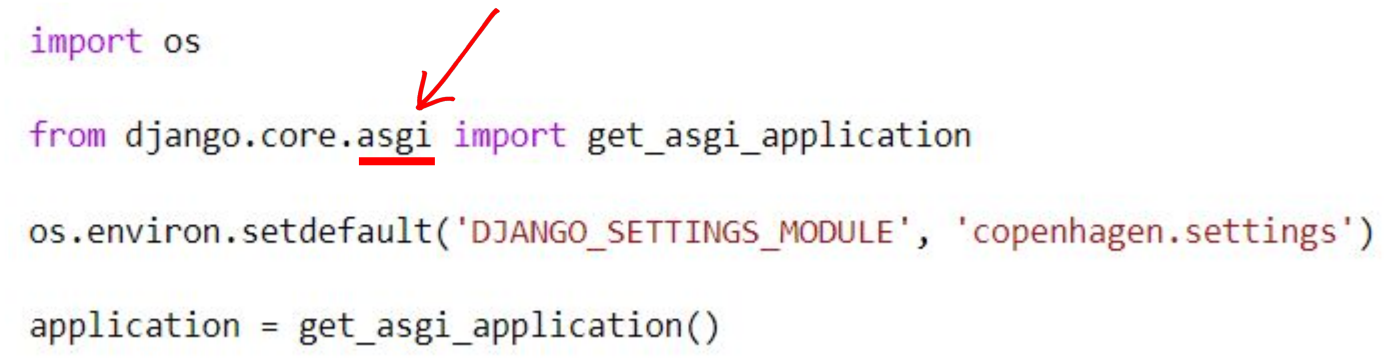
- "spiritual successor to WSGI", compatible with WSGI
- async/await operation support
- websockets
- HTTP and HTTP/2 protocols

ASGI



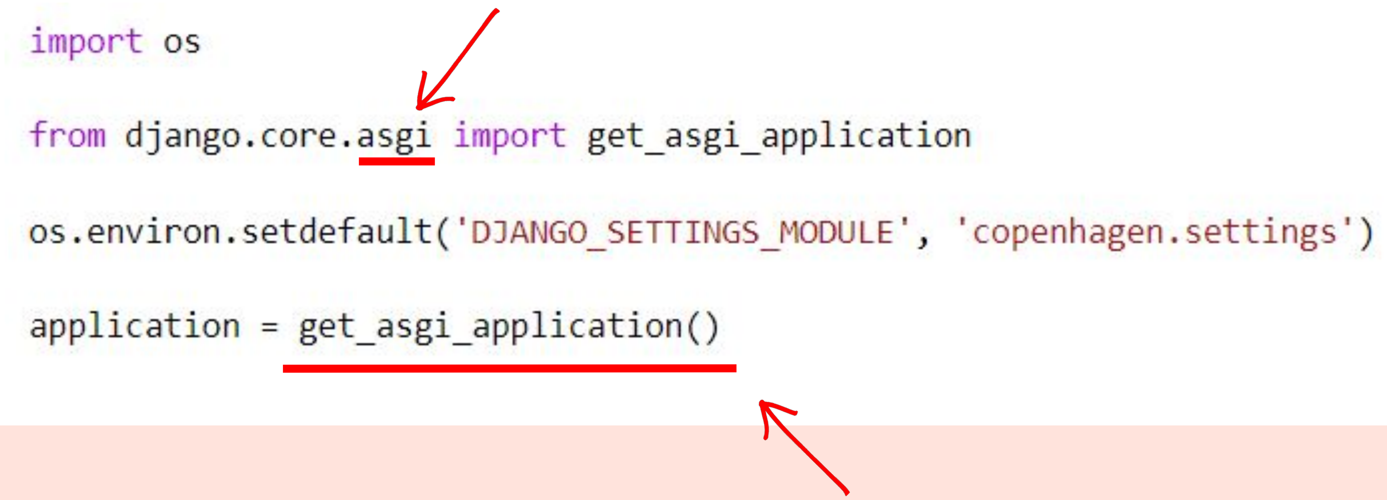
```python
import os

from django.core.asgi import get_asgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'copenhagen.settings')

application = get_asgi_application()
```

ASGI



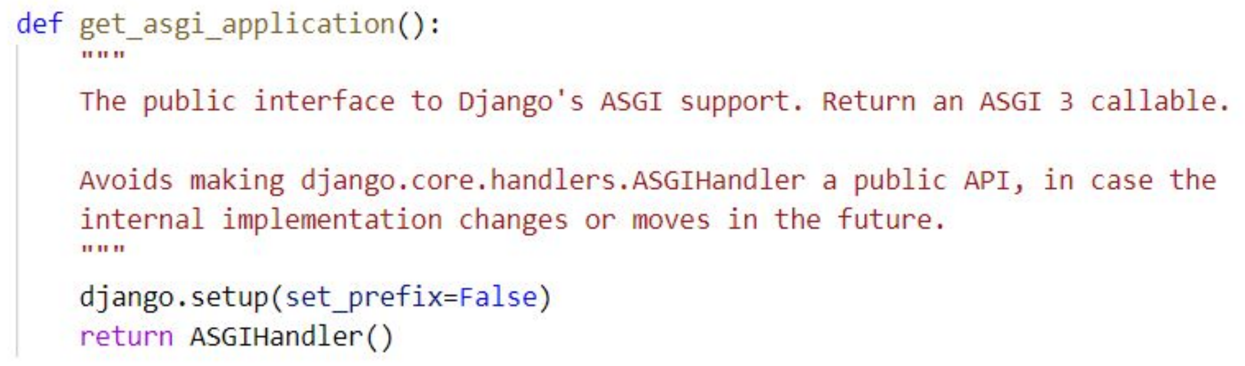
```
import os

from django.core.asgi import get_asgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'copenhagen.settings')

application = get_asgi_application()
```

ASGI



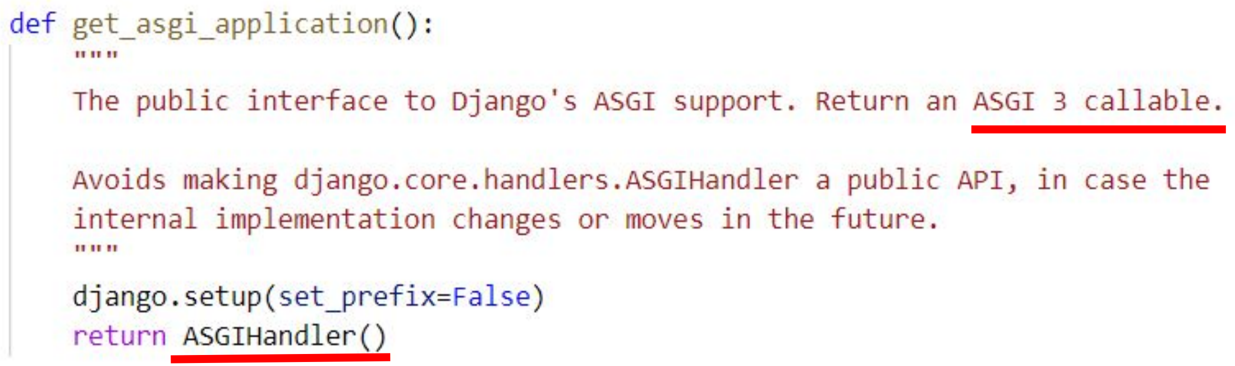
```python
import os

from django.core.asgi import get_asgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'copenhagen.settings')

application = get_asgi_application()
```

ASGI



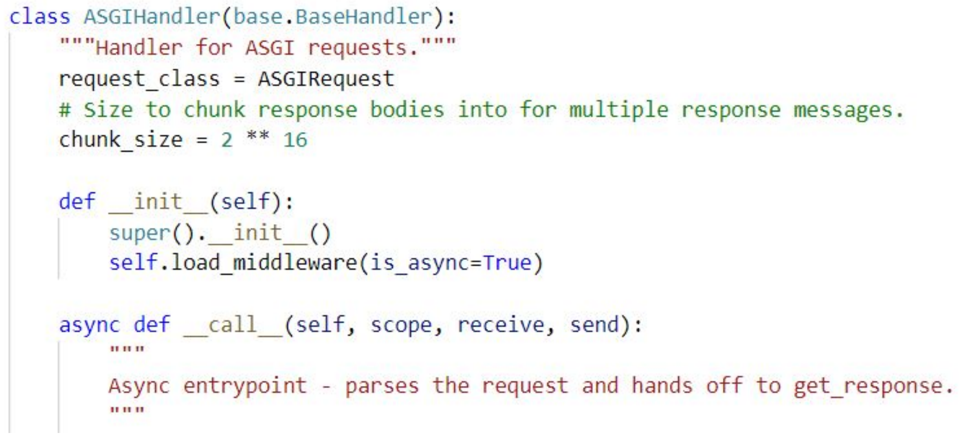
```python
def get_asgi_application():
    """
    The public interface to Django's ASGI support. Return an ASGI 3 callable.

    Avoids making django.core.handlers.ASGIHandler a public API, in case the
    internal implementation changes or moves in the future.
    """
    django.setup(set_prefix=False)
    return ASGIHandler()
```

ASGI



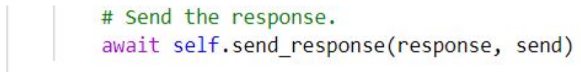
```
def get_asgi_application():
    """
    The public interface to Django's ASGI support. Return an ASGI 3 callable.

    Avoids making django.core.handlers.ASGIHandler a public API, in case the
    internal implementation changes or moves in the future.
    """
    django.setup(set_prefix=False)
    return ASGIHandler()
```

ASGI



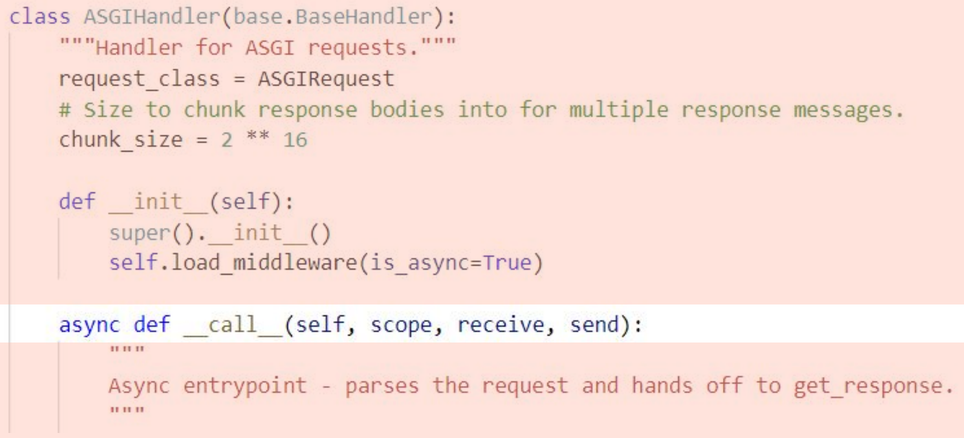
```python
class ASGIHandler(base.BaseHandler):
    """Handler for ASGI requests."""
    request_class = ASGIRequest
    # Size to chunk response bodies into for multiple response messages.
    chunk_size = 2 ** 16

    def __init__(self):
        super().__init__()
        self.load_middleware(is_async=True)

    async def __call__(self, scope, receive, send):
        """
        Async entrypoint - parses the request and hands off to get_response.
        """
```

[....]



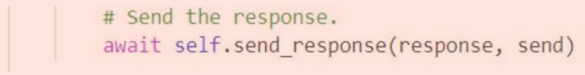
```python
        # Send the response.
        await self.send_response(response, send)
```

ASGI



```python
class ASGIHandler(base.BaseHandler):
    """Handler for ASGI requests."""
    request_class = ASGIRequest
    # Size to chunk response bodies into for multiple response messages.
    chunk_size = 2 ** 16

    def __init__(self):
        super().__init__()
        self.load_middleware(is_async=True)

    async def __call__(self, scope, receive, send):
        """
        Async entrypoint - parses the request and hands off to get_response.
        """
```

[....]



```python
        # Send the response.
        await self.send_response(response, send)
```

ASGI ARGUMENTS

1

scope

- a dictionary with at least a key(`type`) to specify the incoming protocol
- equivalent of `environ` in WSGI

receive

- awaitable callable that will yield an event dictionary

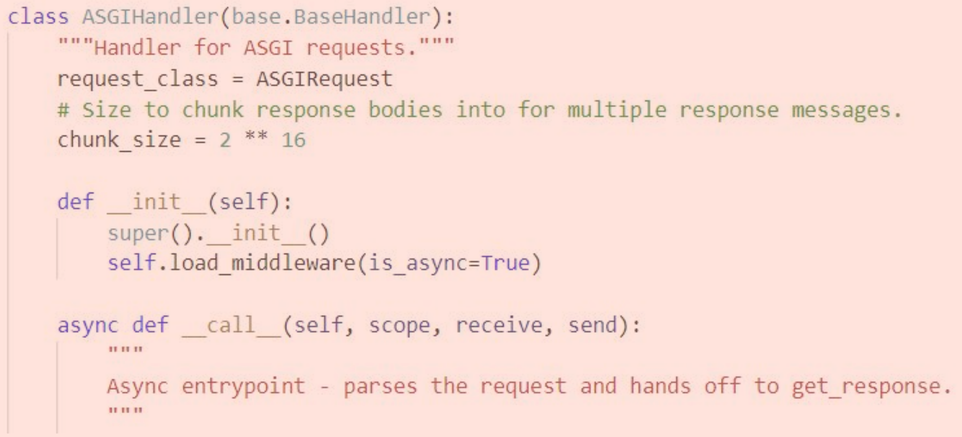
2

3

send

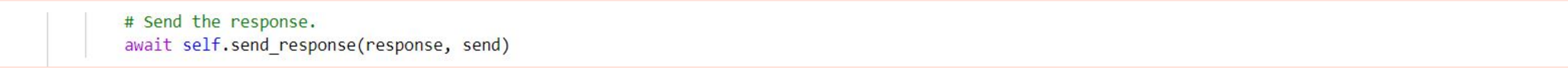
- awaitable callable that takes an event dictionary as a parameter and returns a response once the message has been sent or the connection closed

ASGI



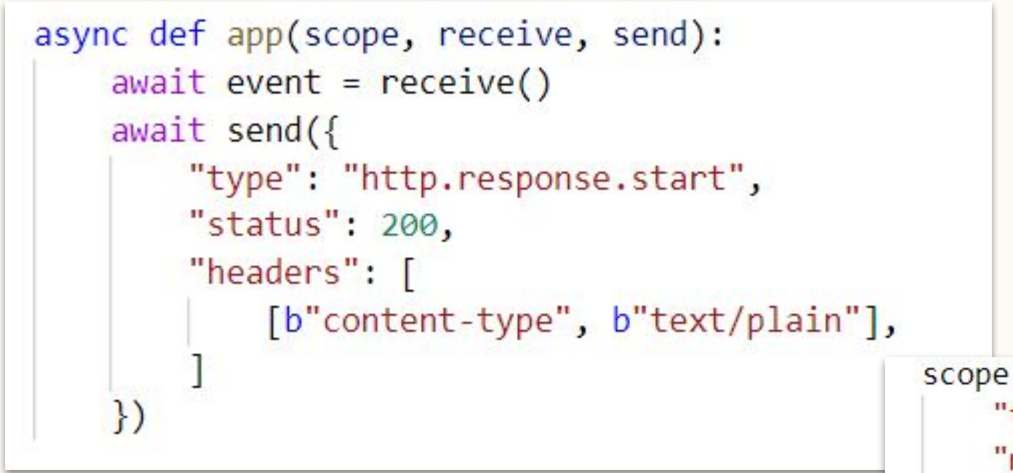
```python
class ASGIHandler(base.BaseHandler):
    """Handler for ASGI requests."""
    request_class = ASGIRequest
    # Size to chunk response bodies into for multiple response messages.
    chunk_size = 2 ** 16

    def __init__(self):
        super().__init__()
        self.load_middleware(is_async=True)

    async def __call__(self, scope, receive, send):
        """
        Async entrypoint - parses the request and hands off to get_response.
        """
```

[....]

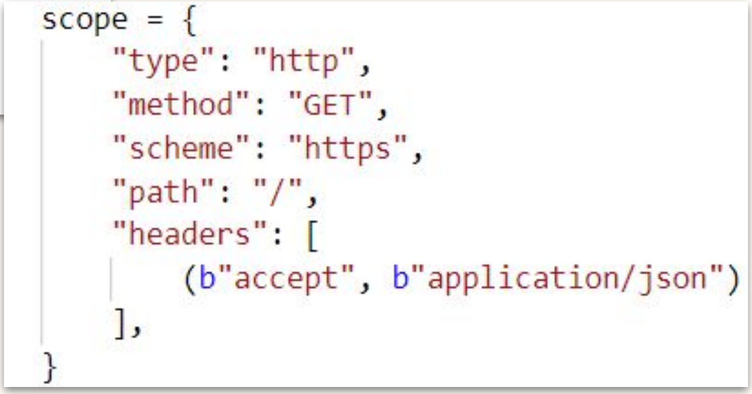


```python
        # Send the response.
        await self.send_response(response, send)
```

ASGI examples

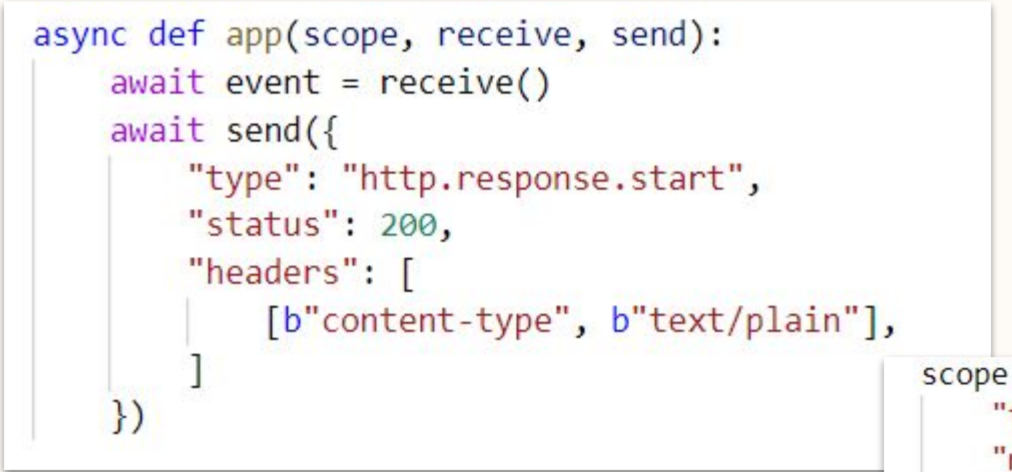


```python
async def app(scope, receive, send):
    await event = receive()
    await send({
        "type": "http.response.start",
        "status": 200,
        "headers": [
            [b"content-type", b"text/plain"],
        ]
    })
```



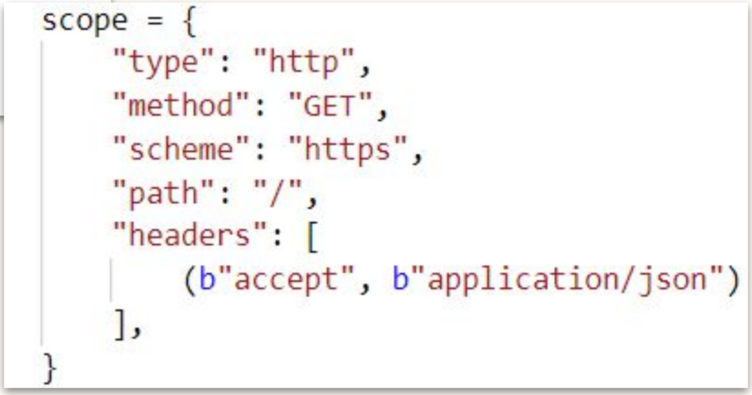
```python
scope = {
    "type": "http",
    "method": "GET",
    "scheme": "https",
    "path": "/",
    "headers": [
        (b"accept", b"application/json")
    ],
}
```

ASGI examples



```
async def app(scope, receive, send):
    await event = receive()
    await send({
        "type": "http.response.start",
        "status": 200,
        "headers": [
            [b"content-type", b"text/plain"],
        ]
    })
```

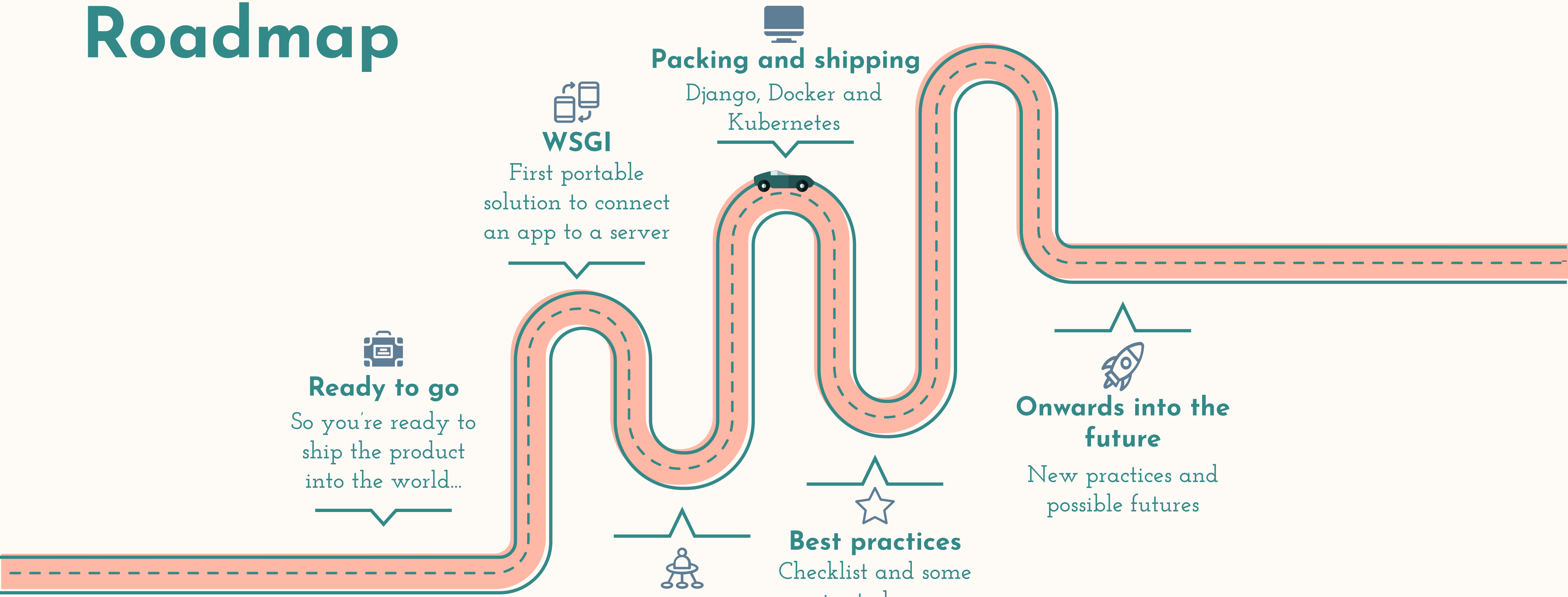
follows the WSGI environ dictionary



```
scope = {
    "type": "http",
    "method": "GET",
    "scheme": "https",
    "path": "/",
    "headers": [
        (b"accept", b"application/json")
    ],
}
```

When can ASGI save the day?

Roadmap



Ready to go
So you're ready to ship the product into the world...

WSGI
First portable solution to connect an app to a server

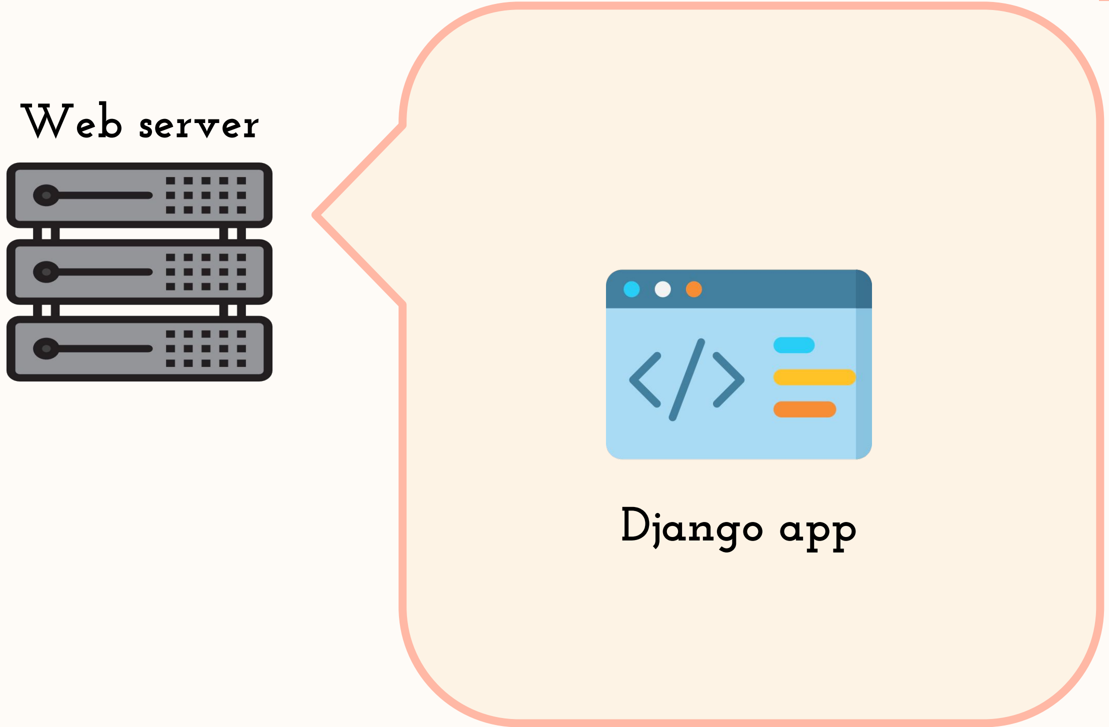
ASGI
The power of async

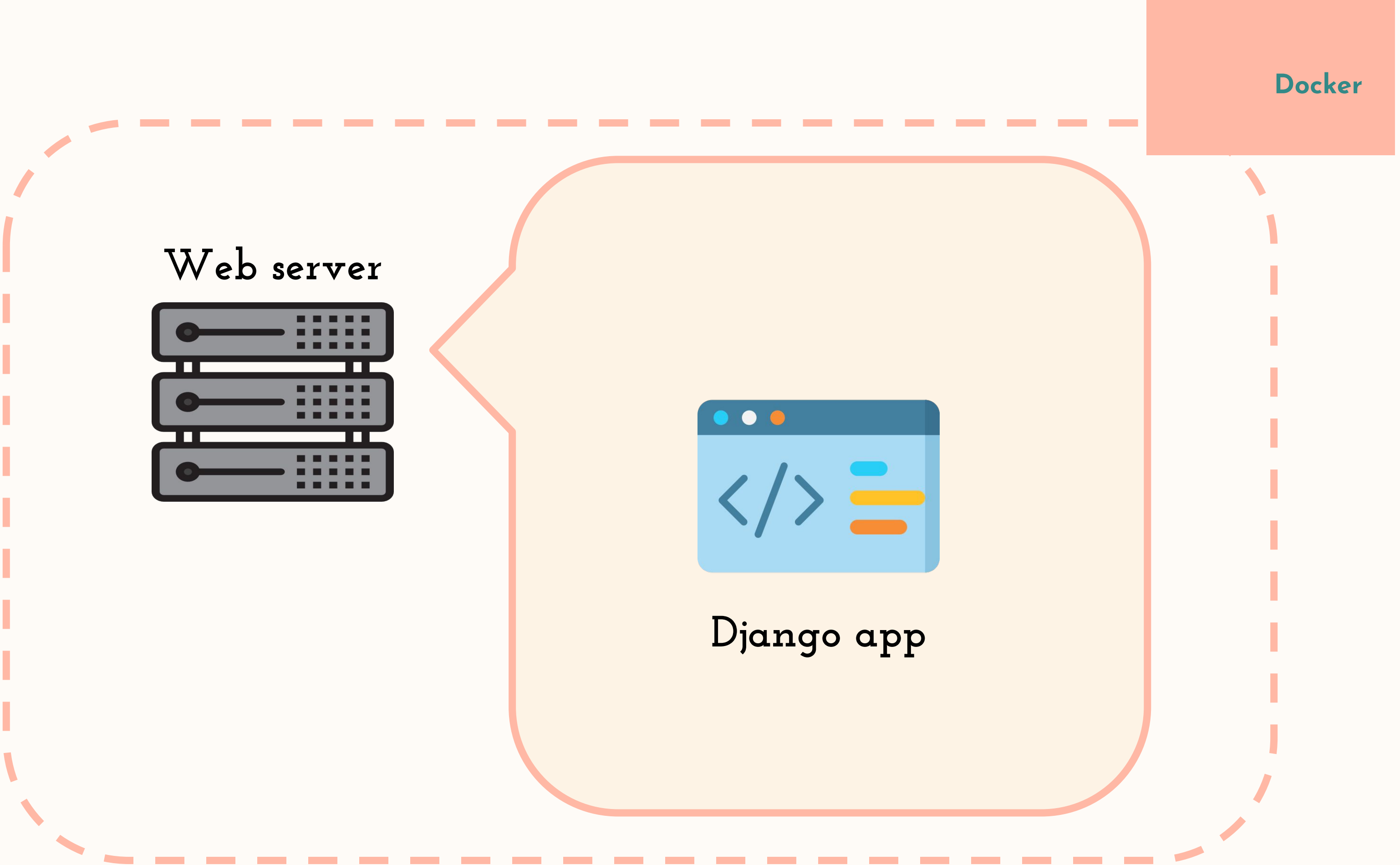
Packing and shipping
Django, Docker and Kubernetes

Best practices
Checklist and some nice to haves

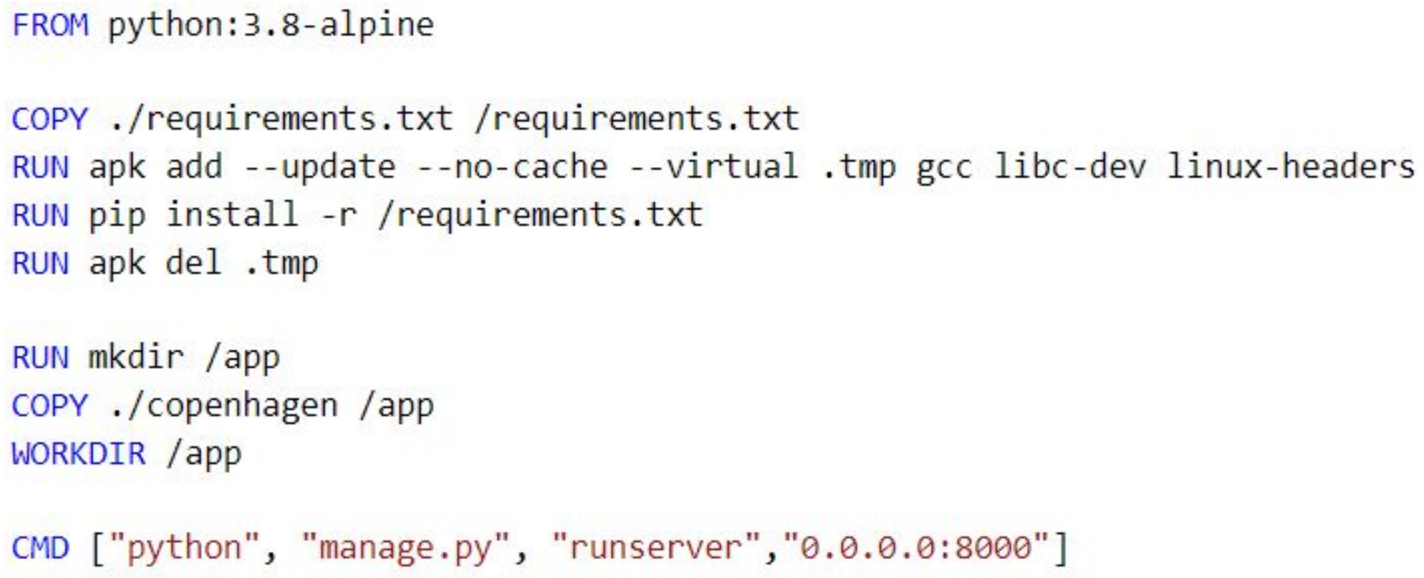
Onwards into the future
New practices and possible futures

Docker




Docker
Web server
Django app

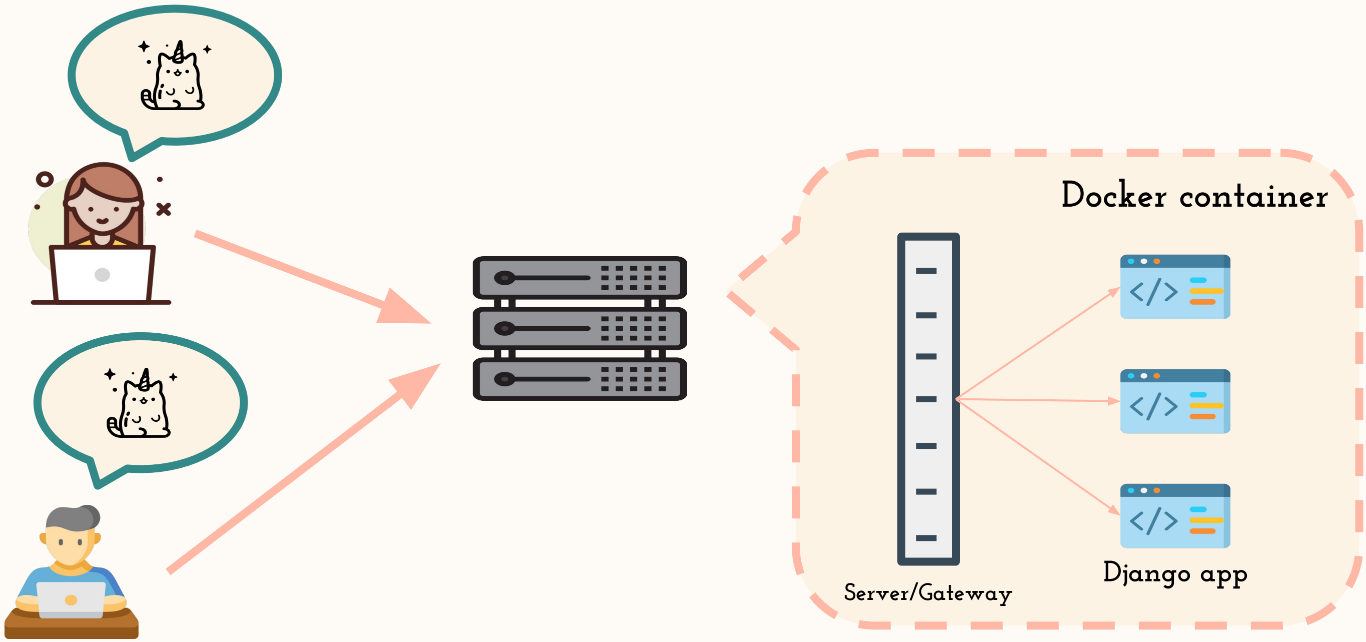
Docker



```dockerfile
FROM python:3.8-alpine

COPY ./requirements.txt /requirements.txt
RUN apk add --update --no-cache --virtual .tmp gcc libc-dev linux-headers
RUN pip install -r /requirements.txt
RUN apk del .tmp

RUN mkdir /app
COPY ./copenhagen /app
WORKDIR /app

CMD ["python", "manage.py", "runserver","0.0.0.0:8000"]
```

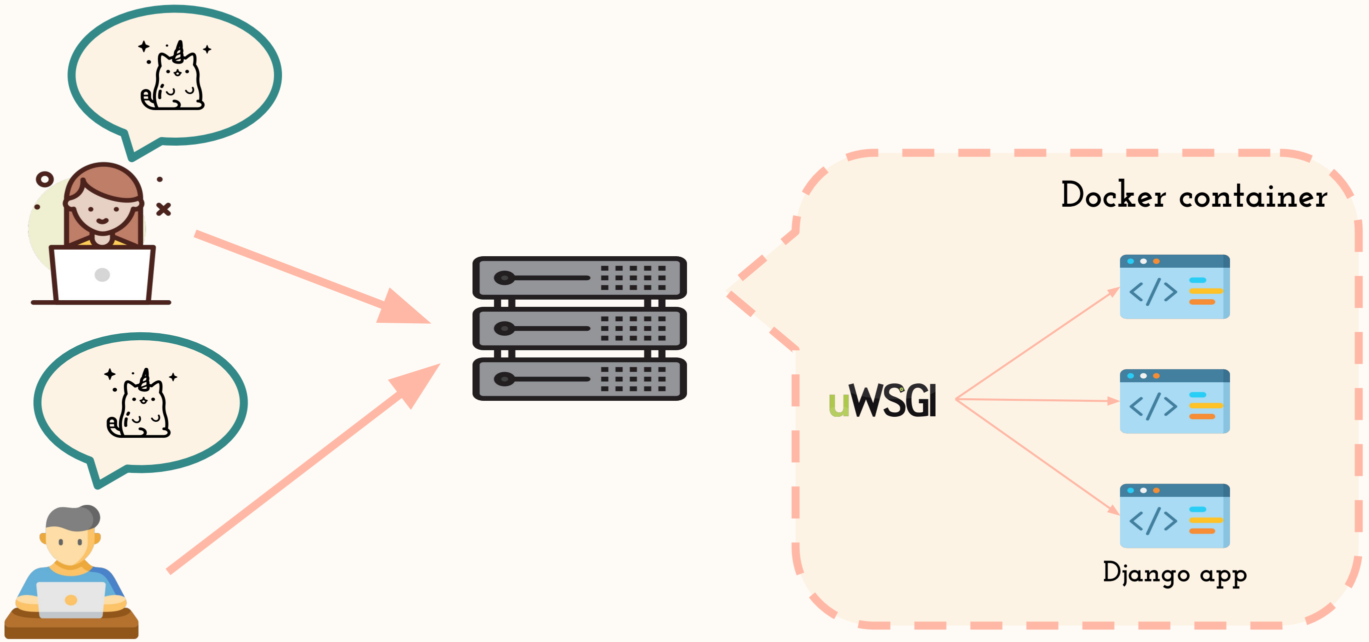
Now let's install the first server on top of our Django application.

This permits us to have multi-threaded operations.



Now let's install the first server on top of our Django application.

This permits us to have multi-threaded operations.



Docker

```
FROM python:3.8-alpine

COPY ./requirements.txt /requirements.txt
RUN apk add --update --no-cache --virtual .tmp gcc libc-dev linux-headers
RUN pip install -r /requirements.txt
RUN apk del .tmp

RUN mkdir /app
COPY ./copenhagen /app
WORKDIR /app
```

~~CMD ["python", "manage.py", "runserver","0.0.0.0:8000"]~~

```
CMD ["uwsgi", "--ini", "uwsgi.ini"]
```

Docker

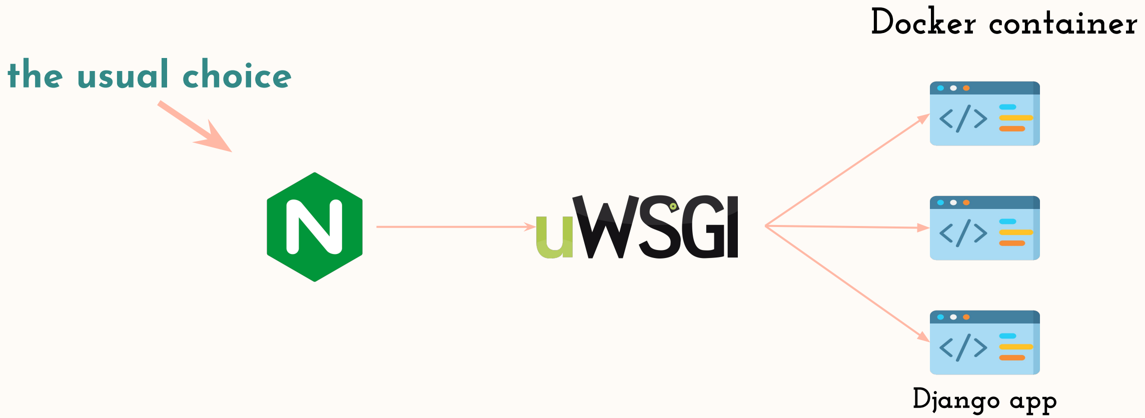
```
FROM python:3.8-alpine

COPY ./requirements.txt /requirements.txt
RUN apk add --update --no-cache --virtual .tmp gcc libc-dev linux-headers
RUN pip install -r /requirements.txt
RUN apk del .tmp

RUN mkdir /app
COPY ./copenhagen /app
WORKDIR /app
```

~~CMD ["python", "manage.py", "runserver","0.0.0.0:8000"]~~

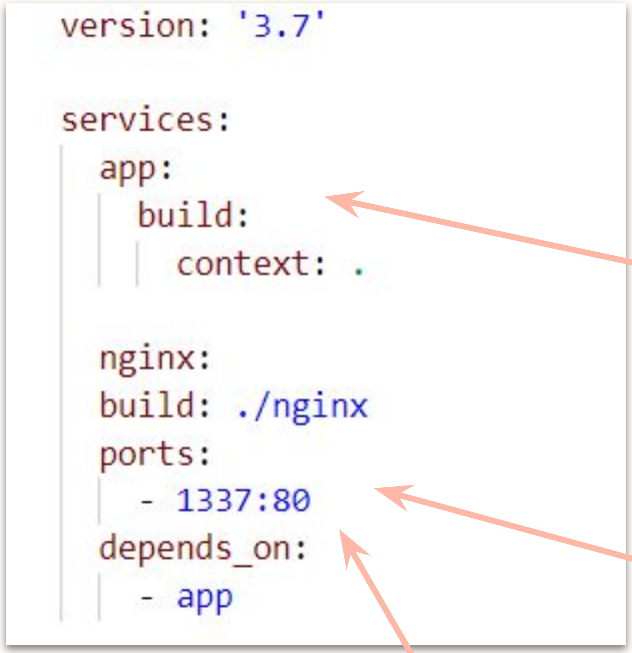
```
CMD ["uwsgi", "--ini", "uwsgi.ini"]
```

- socket
- module
- how many workers
- what to do on exit
- etc

Now usually comes the part where you add another server on top. Or a gateway. Or a load balancer.



docker-compose to the rescue



```
version: '3.7'

services:
  app:
    build:
      context: .

  nginx:
  build: ./nginx
  ports:
    - 1337:80
  depends_on:
    - app
```

build a container for the app accessed by a WSGI/ASGI compliant server (uWsgi earlier)

build a container for the reverse proxy and link it to the app server

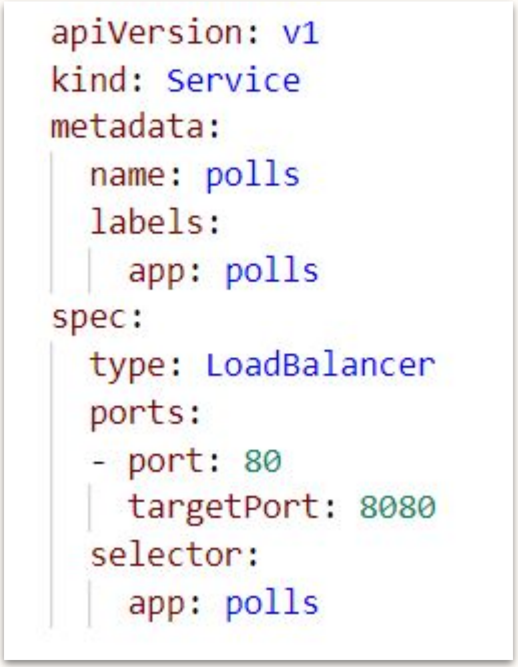
you will need a Dockerfile for it and a file for parameters; and don't forget to touch up STATIC_URL and STATIC_ROOT if you're serving static files

paying attention to port binding can save you a lot of headaches

The next step after that is deploying to some container orchestration tool such as Kubernetes.

- clustering different containers together
- scalable and configurable
- easier deployment and management

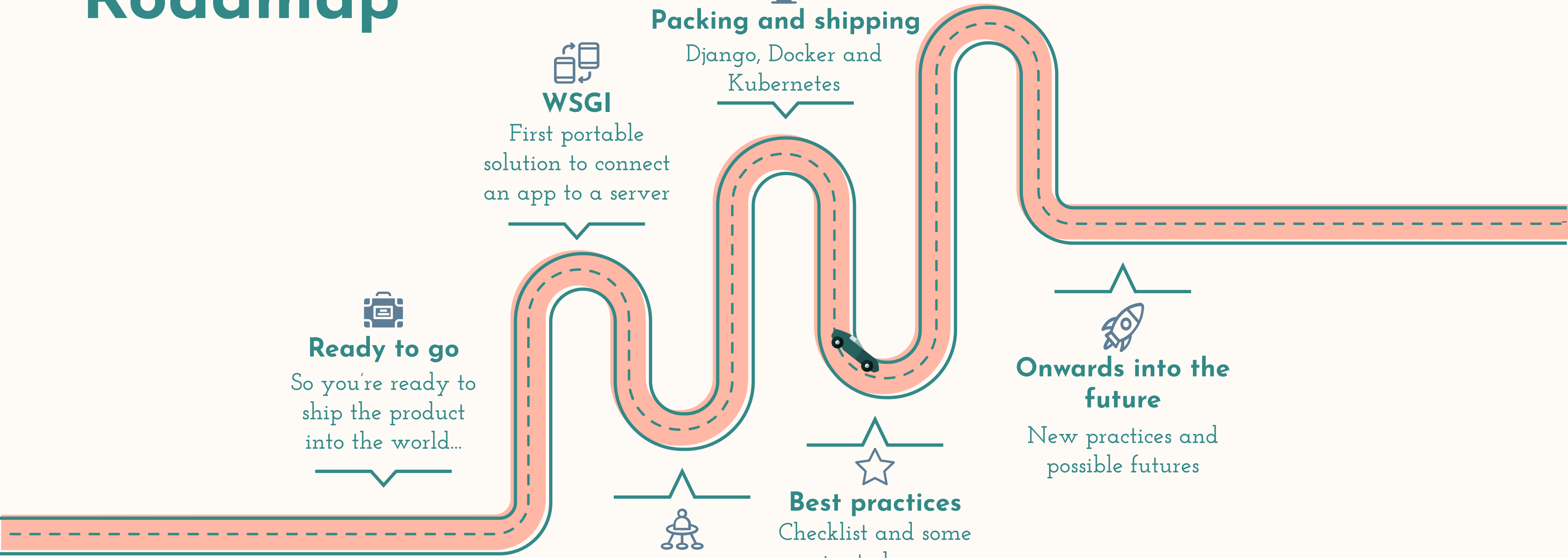
Kubernetes YML example



```yaml
apiVersion: v1
kind: Service
metadata:
  name: polls
  labels:
    app: polls
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: polls
```

Source: https://cloud.google.com/python/django/kubernetes-engine

Roadmap



Ready to go

So you're ready to ship the product into the world...

WSGI

First portable solution to connect an app to a server

ASGI

The power of async

Packing and shipping

Django, Docker and Kubernetes

Best practices

Checklist and some nice to haves

Onwards into the future

New practices and possible futures

Some best practices I learned over time

(sometimes the hard way)

Some best practices I learned over time

(sometimes the hard way)

Use the checklist

The Django checklist is very useful and it is recommended that you use it when deploying. Add items to the checklist that suit your needs.

Some best practices I learned over time

(sometimes the hard way)

Use the checklist

The Django checklist is very useful and it is recommended that you use it when deploying. Add items to the checklist that suit your needs.

Monitor

Don't forget to log. And read those logs. Use the tools available.

Some best practices I learned over time

(sometimes the hard way)

Be careful of sensitive data

Take care of your users. Use environment variables where possible. Act apprehensive when it comes to security.

Use the checklist

The Django checklist is very useful and it is recommended that you use it when deploying. Add items to the checklist that suit your needs.

Monitor

Don't forget to log. And read those logs. Use the tools available.

Some best practices I learned over time

(sometimes the hard way)

Be careful of sensitive data

Take care of your users. Use environment variables where possible. Act apprehensive when it comes to security.

Keep Docker files clean

The order in which you run commands matters. Don't give root permissions to the server.

Use the checklist

The Django checklist is very useful and it is recommended that you use it when deploying. Add items to the checklist that suit your needs.

Monitor

Don't forget to log. And read those logs. Use the tools available.

Some best practices I learned over time

(sometimes the hard way)

Be careful of sensitive data

Take care of your users. Use environment variables where possible. Act apprehensive when it comes to security.

Keep Docker files clean

The order in which you run commands matters. Don't give root permissions to the server.

Use the checklist

The Django checklist is very useful and it is recommended that you use it when deploying. Add items to the checklist that suit your needs.

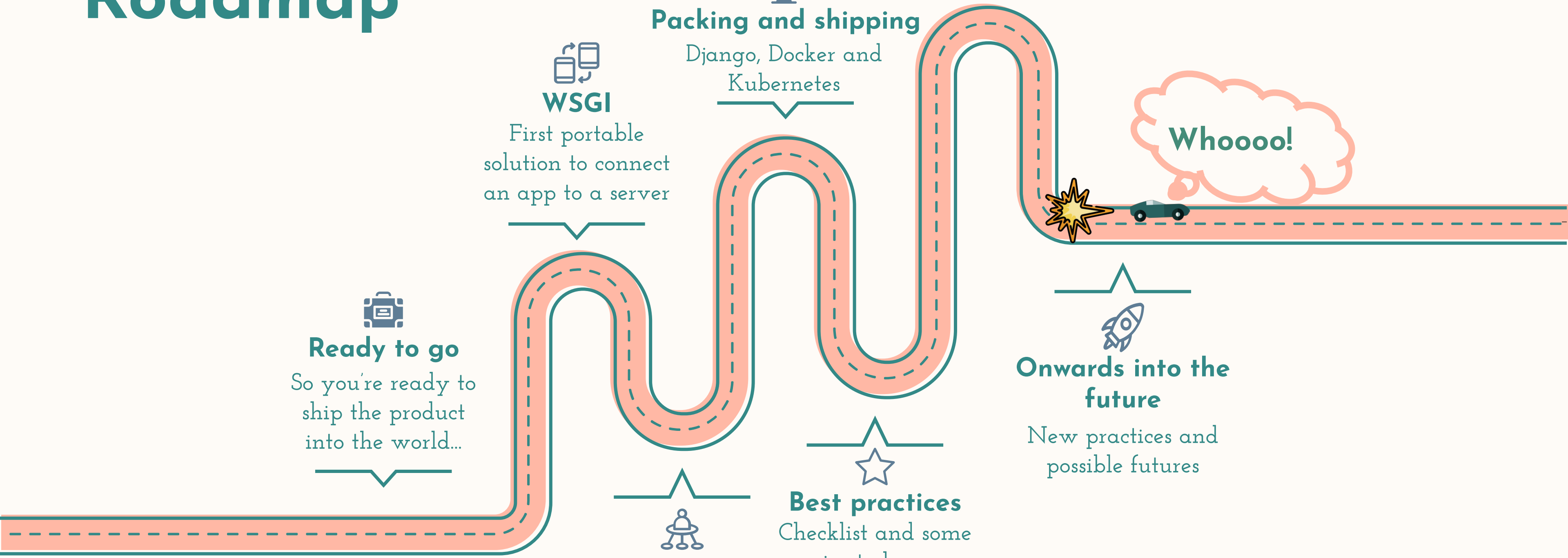
Monitor

Don't forget to log. And read those logs. Use the tools available.

Define what you want and stick to it

You are in control of which tools or patterns you're going to use. Mix and match. If something doesn't work, change it.

Roadmap



Ready to go

So you're ready to ship the product into the world...

WSGI

First portable solution to connect an app to a server

ASGI

The power of async

Packing and shipping

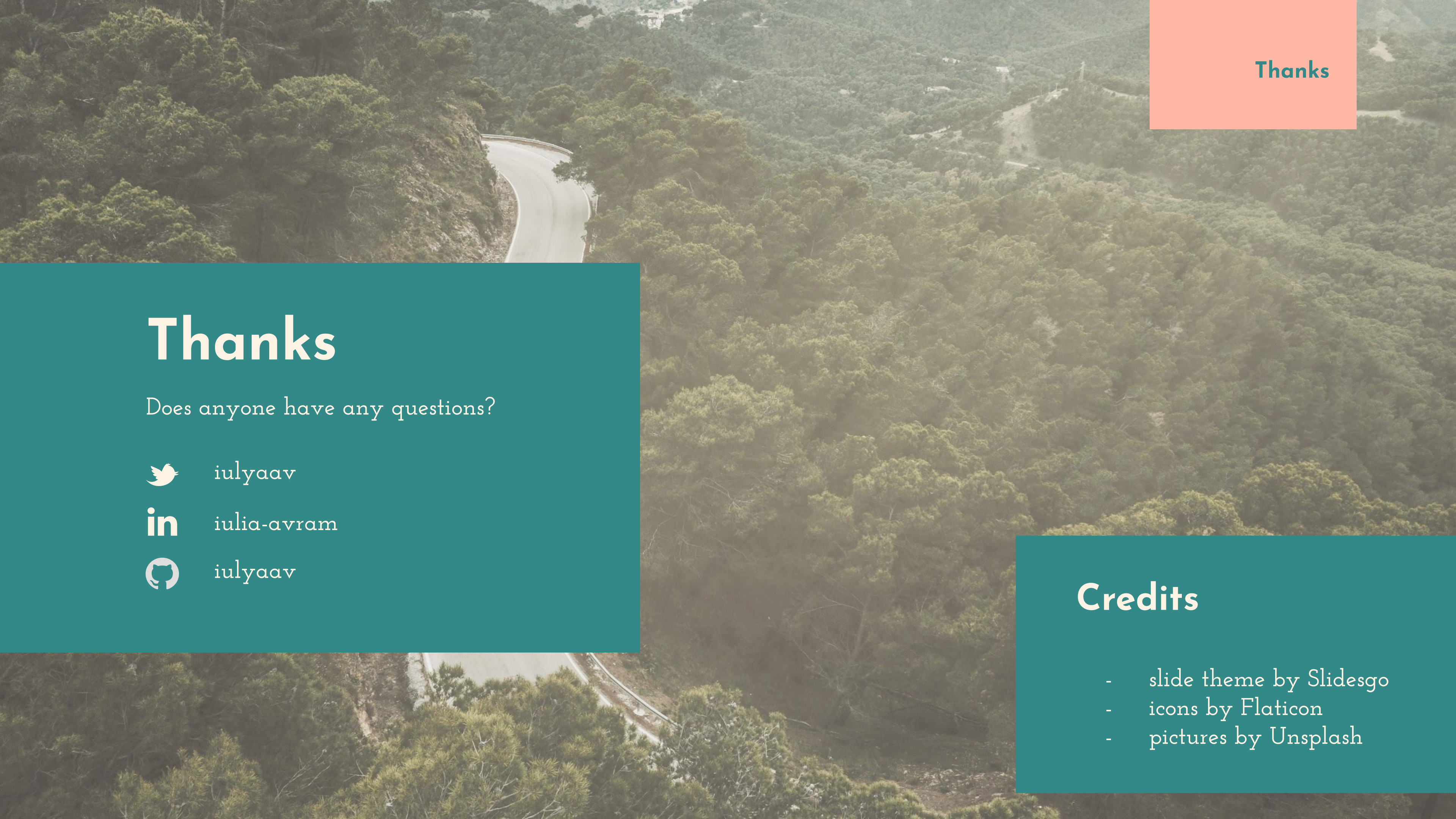
Django, Docker and Kubernetes

Best practices

Checklist and some nice to haves

Onwards into the future

New practices and possible futures



Thanks

Does anyone have any questions?

iulyaav

iulia-avram

iulyaav

Credits

- slide theme by Slidesgo
- icons by Flaticon
- pictures by Unsplash