SWPP Practice Session #10

Optimization & Deployment

Database Optimization

Database optimization

- Why important?
  - Because the database hit is expensive
- Misusing your database queries makes your app slow
  - Scraping whole table
  - Redundant, repetitive queries
  - etc

Do not evaluate the whole `QuerySet`

- Django's `QuerySet` is lazy
- `QuerySet` is evaluated when
  - Iterating, Slicing
  - Pickling, Caching
  - `len()`, `repr()`, `list()`
- Evaluating large `QuerySet` will slow down your whole database actions

Example project

- Create a simple table and migrate
- … and create 10,000 items

```python
from django.db import models


class Item(models.Model):
    name = models.CharField(max_length=30)
    stock = models.PositiveIntegerField(default=0)
```

```python
print('Generating 10,000 random Item models...')

charset = string.ascii_letters + string.digits
for i in range(10000):
    random_name = ''.join(random.choices(charset, k=20))
    random_stock = random.randrange(1024)
    Item.objects.create(name=random_name, stock=random_stock)
    print(f'Created {random_name} item')

print('Done!')
```

Test it

- Write a test script that counts the table entries
  - Calling `len()` and `.count()`

```
8  from app.models import Item
9  from time import time
10
11 start = time()
12 count = len(Item.objects.all())
13 print('Not optimal: {:3f} ms'.format((time() - start) * 1000))
14
15 start = time()
16 count = Item.objects.count()
17 print('Optimal: {:3f} ms'.format((time() - start) * 1000))
```

Test result

- `len()` is about 100x slower than `.count()`
  - This will be gone more serious when your database grows

```
swpp ❯ python count_table_entry.py
Not optimal: 89.265823 ms
Optimal: 0.873804 ms
```

Use appropriate field type

- Keep your database small
  - name is enough to be `CharField`, not `TextField`
- `DateTimeField` vs `DateField`
- `IntegerField` vs `SmallIntegerField` vs `BigIntegerField`
- https://docs.djangoproject.com/en/2.1/ref/models/fields/

Use `.values()` and `.values_list()`

- Don’t retrieve the fields that you do not need
  - Reduce the database overhead
  - Retrieve as Python `dict` or `list`, not model object itself

```
>>> from app.models import Item
>>>
>>> item = Item.objects.filter(id=1).values('name')[0]
>>>
>>> item
{'name': 'hERDKDozPIK3RH3C9bQM'}
>>>
```

Use ForeignKey value directly

- If you know the ForeignKey's primary key(id), you can use it directly
  - Append '`_id`' to the field name
  - More efficient due to reduced queries

```
>>> author = Author.objects.get(id=1)
>>>
>>> new_book = Book.objects.create(name='Hello', author=author)
>>>
```

vs

```
>>> new_book = Book.objects.create(name='World', author_id=1)
>>>
```

Use DB index

- DB index can boost your query speed, such as `filter()`
- Consider adding DB index into the fields that you frequently use `filter()`, `exclude()`, `order_by()`, etc

```
class Item(models.Model):
    name = models.CharField(max_length=30, db_index=True)
    stock = models.PositiveIntegerField(default=0)
```

Use Transaction

- Enclose several queries in a transaction
  - Commit at once
  - Rollback to initial state when exceptions was raised
- Transaction will help you to keep your database state integrity

```python
from django.db import transaction

@transaction.atomic
def viewfunc(request):
    # This code executes inside a transaction.
    do_stuff()
```

```python
from django.db import transaction

def viewfunc(request):
    # This code executes in autocommit mode (Django's default).
    do_stuff()

    with transaction.atomic():
        # This code executes inside a transaction.
        do_more_stuff()
```

Understand cached attributes

- Once your QuerySet or model object is evaluated, its attributes was cached
  - This is same in terms of related object
    - `ForeignKey`, `ManyToManyFields`

```
>>> from app.models import Item
>>> item = Item.objects.get(id=1)
>>> item.name
'hERDKDozPIK3RH3C9bQM'
>>> item.stock
519
>>>
```

Hit the database

Use cached attribute

Use cached attribute

Retrieve related fields in one query

- Example tables

```
class Author(models.Model):
    name = models.CharField(max_length=30)


class Book(models.Model):
    name = models.CharField(max_length=30)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
```

Retrieve related fields in one query

- Query a Book object and lookup its name and author's name

```
>>> from app.models import Book, Author
>>> book = Book.objects.get(id=1)
>>> book.name
'SWPP'
>>>
>>> book.author.name
'skystar'
```

Hit the database

Use cached attribute

Hits the database again, No!

Any solution?

Use `select_related()`

- Prepopulate the related field in one query
  - Use `prefetch_related()` similarly in `ManyToManyFields`

```
>>> from app.models import Book, Author
>>> book = Book.objects.select_related('author').get(id=1)
>>>
>>> book.name
'SWPP'
>>>
>>> book.author.name
'skystar'
>>>
```

First hit

Use cached attribute

No database hit, yes!

Efficient!

Use another database backend

- SQLite is easy to use and manage, but
  - Slow
  - Fail at large number of simultaneous connections
- Use battle-proven Database backends
  - PostgreSQL
  - MySQL
  - etc

Conclusion

- Profile and inspect raw SQL queries when you needed
- Proper use of DB query will boost your whole application speed

- https://docs.djangoproject.com/en/2.1/topics/db/optimization/

Cache

Why cache?

- Database is basically file-based storage
  - Disk IO is desperately slow than DRAM access
- Solution?
  - Use memory-based cache to reduce access time
  - Use cache in frequently used, or transient data
    - Session
    - Game state
    - etc

Cache backends

- Local memory cache (default)
  - Fastest
- Database cache
  - If you have fast and well-indexed database server
  - Persistent
- External cache backends
  - Redis
  - Memcached
  - etc

Setting up cache backend

- CACHES setting in your root configuration file
- https://docs.djangoproject.com/en/2.1/topics/cache/

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake',
    }
}
```

Cache the view

- Use `cache_page` decorator
  - This will cache your view per-url
  - Set timeout at decorator parameter

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)
def my_view(request):
    ...
```

15 mins

Use Redis as your cache backend

- Redis is a fast, scalable and versatile in-memory database
  - https://redis.io/topics/introduction
- No native support of Django, so use adapter
  - https://github.com/niwinz/django-redis
- Set your cache backend as `django-redis`

```
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/0',
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
        }
    }
}
```

Use cache as session engine

- Session data is frequently used
  - Storing and reading session data with database is not that good idea
- Use cache based session engine
  - Set at your root configuration file
- Redis also persist your data, so your session data will be remained even after you restart the server
- https://docs.djangoproject.com/ko/2.1/topics/http/sessions/#using-cached-sessions

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
SESSION_CACHE_ALIAS = 'default'
```

Accessing the raw cache

- Import `django.core.cache.cache`
- You can also set expire of cache entry
  - `cache.set(key, value, timeout=<timeout>)`
- Cleverly use to make fast your application
  - But be careful of simultaneous access, always (data races)

```
>>> from django.core.cache import cache
>>>
>>> cache.set('some_key', 'SWPP')
>>>
>>> cache.get('some_key')
'SWPP'
>>>
```

Load Test

Load Testing

- Kind of profiling
- Figure out how many concurrent user a system can handle
- Attack your application and find bottlenecks

Locust

- Scalable user load testing tool
- https://github.com/locustio/locust
  - `$ pip install locust`

Write locust file

- Define tasks to perform
- Several settings

locustfile.py

```
from locust import HttpLocust, TaskSet, task


class LoadTask(TaskSet):
    def on_start(self):
        pass

    def on_stop(self):
        pass

    @task
    def index_page(self):
        self.client.get('/')


class WebsiteUser(HttpLocust):
    task_set = LoadTask
    min_wait = 3000
    max_wait = 8000
```

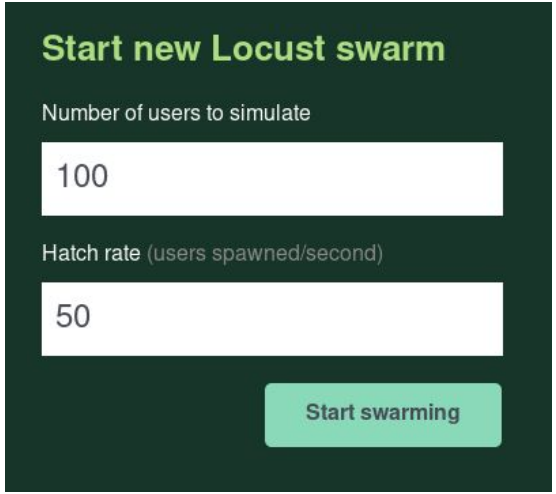
Example view function

- Create a new `Item` object for each request

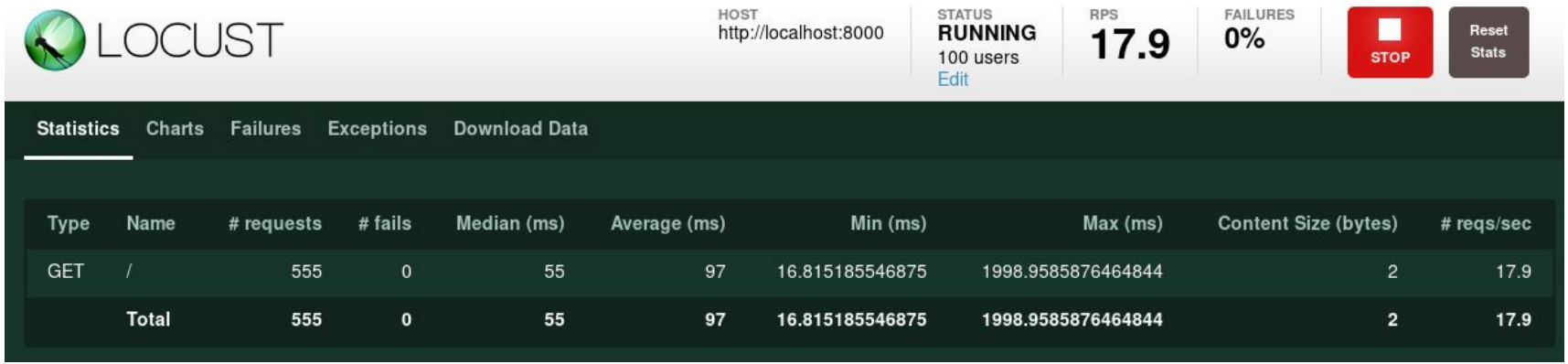
```
def create_item(request):
    if request.method == 'GET':
        charset = string.ascii_letters + string.digits
        random_name = ''.join(random.choices(charset, k=20))

        # create a item of random name
        Item.objects.create(name=random_name)

        return HttpResponse('OK')
    else:
        return HttpResponseNotAllowed(['GET'])
```

Run Locust

- `$ python manage.py runserver`
- `$ locust -f locustfile.py --host=‘http://localhost:8000’`
- Locust web ui will be appear

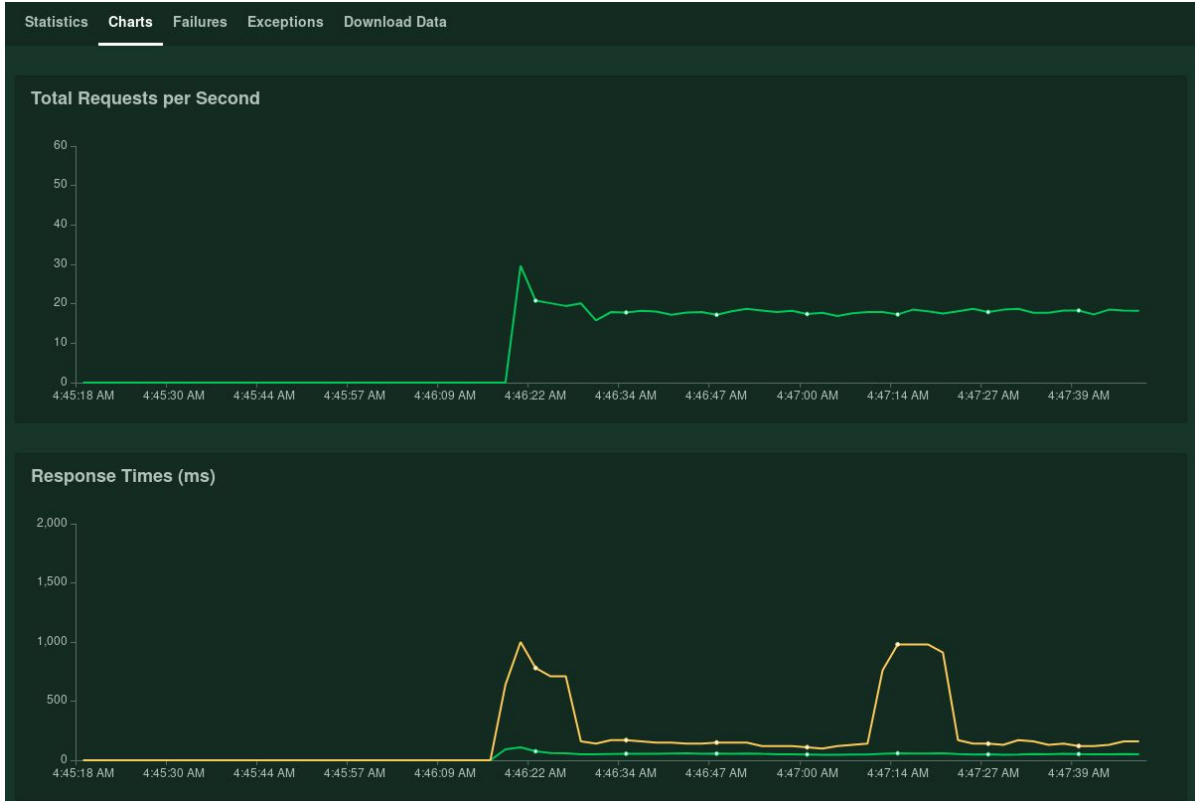


Attack your application



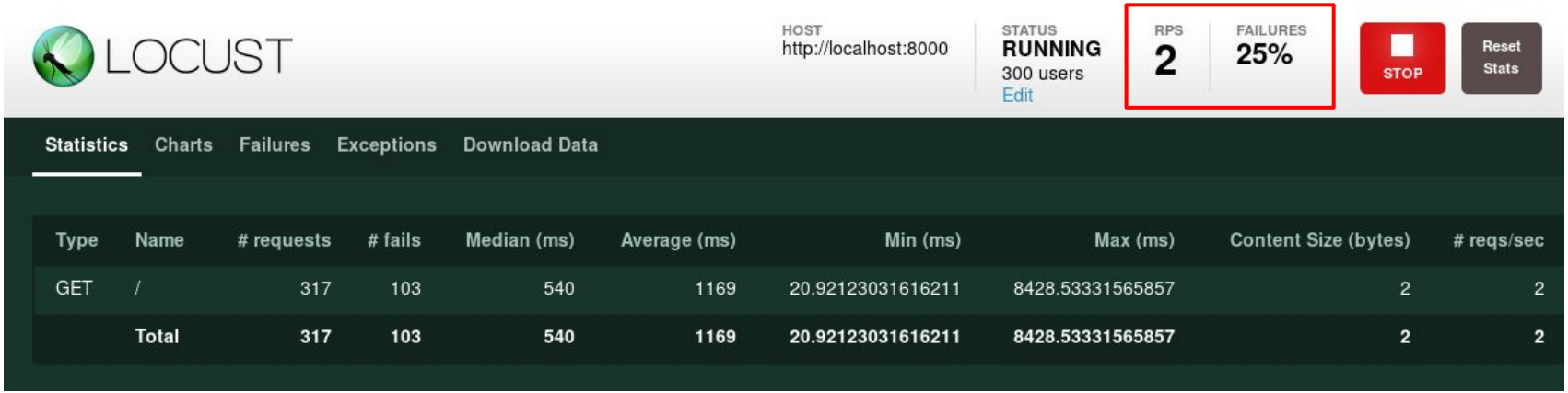
Type	Name	# requests	# fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Content Size (bytes)	# reqs/sec
GET	/	555	0	55	97	16.815185546875	1998.9585876464844	2	17.9
	Total	**555**	**0**	**55**	**97**	**16.815185546875**	**1998.9585876464844**	**2**	**17.9**

Attack your application



Heavy load

- 300 concurrent users



Type	Name	# requests	# fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Content Size (bytes)	# reqs/sec
GET	/	317	103	540	1169	20.92123031616211	8428.53331565857	2	2
	Total	**317**	**103**	**540**	**1169**	**20.92123031616211**	**8428.53331565857**	**2**	**2**

What’s going on?

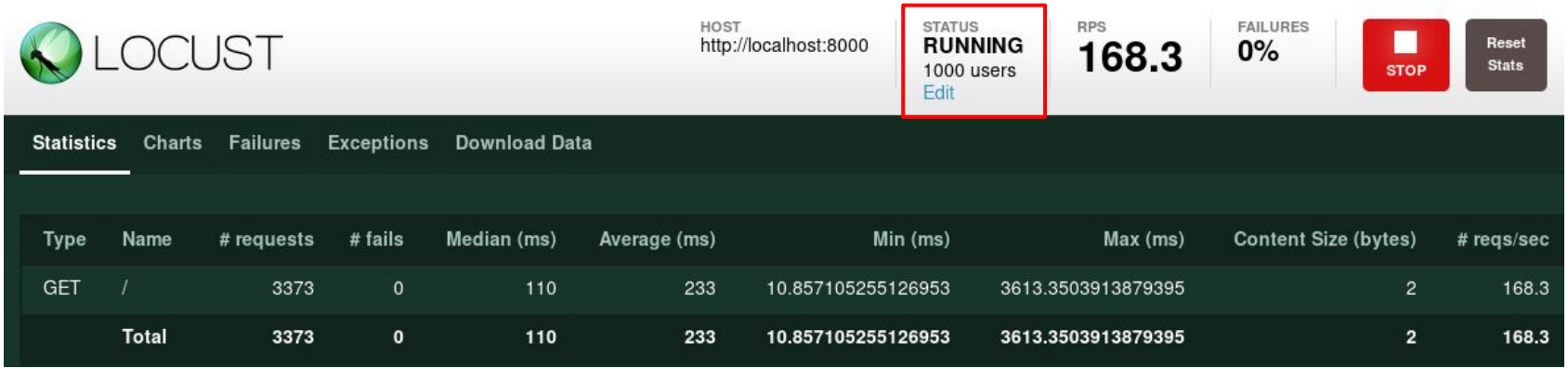
Inspection

- `django.db.utils.OperationalError: database is locked`
- SQLite is not good at handling concurrent requests
  - Use another database backend

```
django.db.utils.OperationalError: database is locked
[23/Nov/2018 19:50:52] "GET / HTTP/1.1" 500 155054
```

Change database backend

- Changed database backend from SQLite to PostgreSQL
  - No problem with 1000 concurrent users!



Type	Name	# requests	# fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Content Size (bytes)	# reqs/sec
GET	/	3373	0	110	233	10.857105255126953	3613.3503913879395	2	168.3
	Total	**3373**	**0**	**110**	**233**	**10.857105255126953**	**3613.3503913879395**	**2**	**168.3**

Angular Optimization

- Performance: https://github.com/mgechev/angular-performance-checklist
  - Lazy Loading
  - Server Side Rendering (Angular Universal)
  - Service Workers (Progressive Web Apps)
  - ...
- Optimization Goal
  - Minimize the time of showing meaningful response
    ex) Load everything then render vs Minimal render first, then fetch rest
  - https://developers.google.com/web/tools/lighthouse/?hl=ko

Deployment

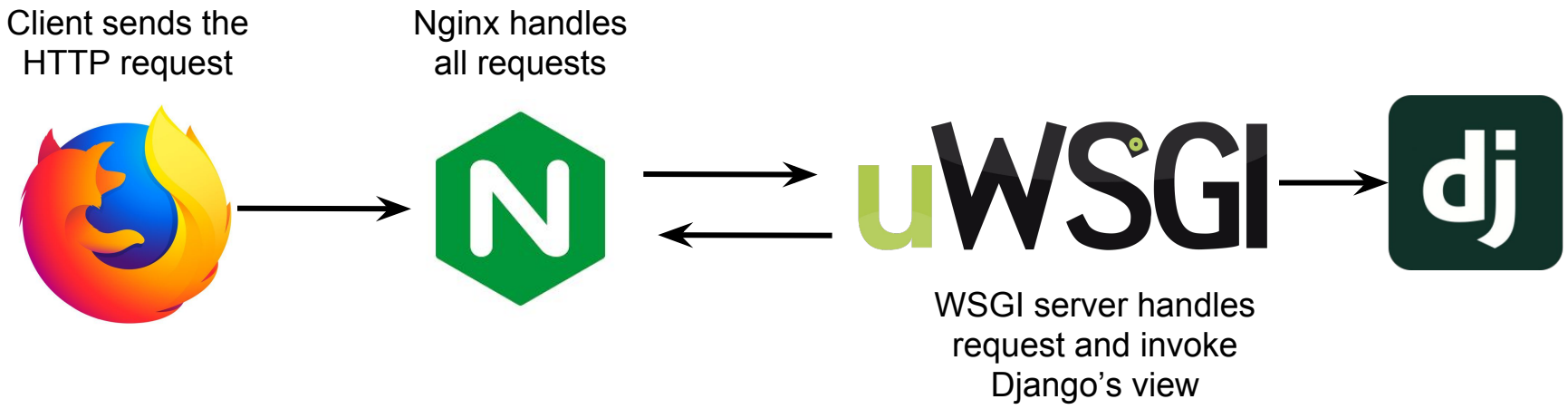
Deployment

- Deployment is not as same as Development
  - Real service vs Testing environment
- Needs to be reliable
  - Self-healing
  - No downtime
  - Survive from harsh load
- Not sufficient with development server
  - Do not deploy with `$ python manage.py runserver`

Prepare your server

- Great to go with AWS EC2 instance
  - … or your own servers
- Check the previous slides to know how to launch EC2 instances
- Get your own domain (optional)

Deployment Layout Example



Nginx

- Fast and scalable HTTP & reverse proxy server
  - Built-in caching
  - Fault tolerant
  - Load balancing feature
- Receive all requests from outside, and multiplex them to inner processes
  - Reverse-proxy
- Also supports uwsgi
- https://nginx.org/en/docs/

WSGI

- Web Server Gateway Interface
- Interface for communicating between web server and application
- Django’s primary deployment platform
- Many WSGI servers available
  - uWSGI
    - https://uwsgi-docs.readthedocs.io/en/latest/
  - Gunicorn
    - https://gunicorn.org/
  - Bjoern
    - https://github.com/jonashaag/bjoern

Angular production build

- `$ ng serve` is not adequate for production
- Just build the project and serve the static files
  - `$ ng build --prod`
  - And serve the output files with Nginx
  - https://docs.nginx.com/nginx/admin-guide/web-server/serving-static-content/
    - Use `try_files` or `root` directive
    - Also read optimization section of above link

HTTPS

- Why?
  - Encrypted
  - Integrity
  - etc
- Grab your certificate-key pair using Let’s Encrypted
  - https://letsencrypt.org/
  - Certbot will help you to easily do this work
    - https://certbot.eff.org/

WebSocket

- Many teams are using WebSocket
- Handling WebSocket requests requires more work
  - Django Channels will work great with your application
    - https://github.com/django/channels
  - Understand how WebSocket requests will be directed to your app
    - https://github.com/django/daphne
  - Nginx also can handle WebSocket requests
  - Use proper channel layer such as Redis for better performance
    - https://github.com/django/channels_redis

Build your architecture yourself

- There are so many options you can choose
  - What reverse-proxy server?
  - What WSGI server?
  - How about load balancing?
  - Multiple servers?
  - etc
- Check out the tutorials and adapt to your environment

Example configuration

- You can see example configuration for uWSGI + Nginx + Django at:
  - https://uwsgi-docs.readthedocs.io/en/latest/tutorials/Django_and_nginx.html
  - Do not simply follow the tutorial to serve your application, but modify and use adequately for your project environment

Announcement

- Finish your running project deployment before Sprint 5
  - Bug report session
- Add response time / throughput analysis on final report
  - Use load testing tool
- Submit demo video on your final report
  - Also be graded
- Good luck with your final exam!