University of New South Wales

COMP3900

Project Report: *localhost*

Group Name: *Undergrads*

Edward Dai	z3415169@unsw.edu.au	z3415169	Developer
Jasper Lowell	z5180332@unsw.edu.au	z5180332	Scrum Master, Developer
Steven Tudo	z3466724@unsw.edu.au	z3466724	Developer
Thomas Lam	z5136612@unsw.edu.au	z5136612	Developer

October 20, 2018

Contents

1 Introduction

1.1 Motivations

> "Sydney has 600,000 bedrooms not being used according to data compiled by EY from 2016 Census estimates."
>
> *Michael Cranston, Australian Financial Review*

The need for an alternative to hotels for short-term property rentals as well as the abundance of spare rooms in homes has led to the innovation of a number of new platforms such as Airbnb and Stayz. These platforms provide hosts with a marketplace to rent their properties of rooms and provide guests with affordable short-term accommodation. One of the major downsides with Airbnb-like platforms, is that the price for the property or room is static and does not change regardless of demand or lack thereof. If the price of a listing is too high on a particular night but would have been purchased by a guest if it was slightly cheaper, the listing would remain vacant. This is a loss for both the host and the prospective guest. A better pricing model is required, one that is regulated by the supply and demand such that both owners and tenants may benefit financially.

1.2 Aim

The aim of the project was to design and implement a fast-paced auction platform for single night stays. Compared to the static pricing model of existing platforms, the dynamic nature of the auction system optimises the occupancy rate of properties. The price of each property item is determined by market supply and demand, potentially improving profitability for hosts and allowing guests to find cheap accommodation on properties with low demand.

Figure 1.2.1 shows theoretical earnings over the course of a week for a traditional fixed-price platform and our *localhost* platform. Earnings are lower on the traditional platform as the price is deemed too steep on days of low demand and the property is left empty. By contrast, the *localhost* allows income for hosts even on days of low demand, albeit at a lower rate. On days with high demand, the price paid may exceed the price that would have been staticly set on traditional platforms. Guests also benefit from this model as they can find bargains on days of low demand.

Due to the fast-paced nature of our auction system, our project focuses on single night stays for guests.

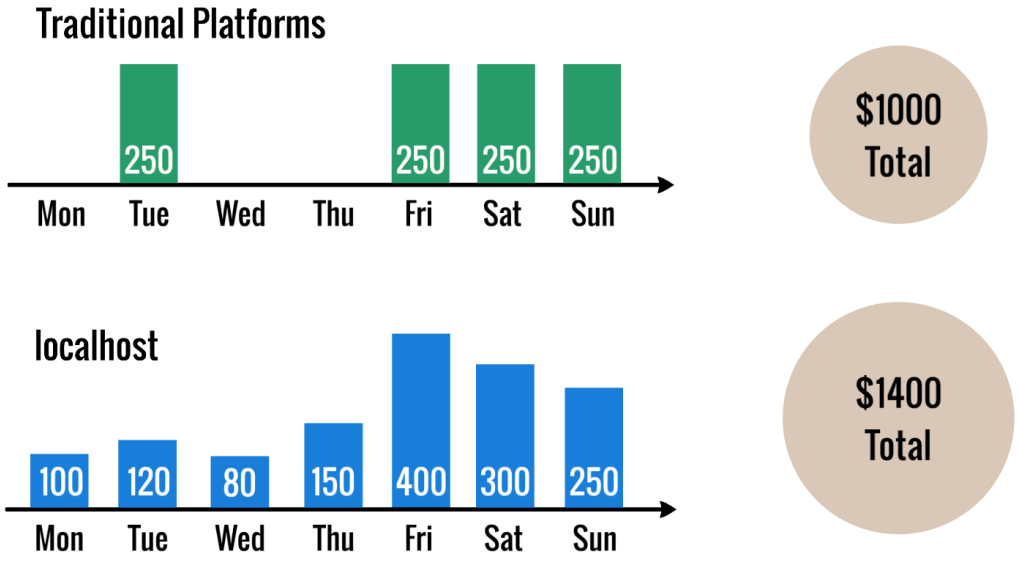

Figure 1.2.1: Theoretical comparison of Fixed-Pricing vs Auctions

1.3 Concepts

1.3.1 Properties and Property Items

As well as listing entire properties, hosts can choose to list individual rooms or couches within a single property. These smaller listings within a property are called property items, and are displayed on each property page.

1.3.2 Auction Sessions

Session	Start Time	End Time
1	12pm	6pm
2	6pm	8pm
3	8pm	10pm
4	10pm	12am
5	12am	3am

Table 1.3.1: Auction Session Times

Guests can book accommodation on *localhost* by winning an auction. Auctions on our platform run in fixed sessions ranging in time from two to six hours. At the end of every session the highest bidder wins the auction and a booking is confirmed. As seen in Table 1.3.1, auction sessions start running from midday to 3am the following day, with shorter sessions for

stages of the day where a higher volume of bids are expected and longer sessions for periods of low volume such as early afternoon.

The purpose of having multiple sessions is to give prospective guests multiple chances to win a bid throughout the day. If the guest loses a bid earlier in the day, he has additional chances later that night to win another bid. If auction sessions were not enforced and owners could choose their own session times, they would choose a single long session throughout the day which would maximise the number of bids to optimise their profits. Instead, hosts must decide which sessions their properties are listed for auction based on personal factors such as latest guest check-in time.

1.3.3 Buyout Option

A buyout option exists for guests to immediately purchase the item without having to wait for the auction session to end. Guests may choose to trade off cost for convenience and hosts receive large financial benefit from buyouts. Buyouts will immediately end the auction session and award the property item to the buyer.

1.3.4 Wallet

User must first deposit credit into their accounts before they can bid in auctions. Once a bid is made in an auction, the amount is immediately deducted from the wallet. This is to prevent users from bidding on multiple properties and encourages users to commit to a single property or room. Credits are immediately returned to the user's wallet if he or she was outbid or the auction was bought out.

2 Dependencies

This project uses the following dependencies.

2.1 Celery

When an auction session comes to an end, the system needs to check if there are any bids under property item. If there are, the system should clean up all the bids associated with the property item and adds a booking to the winner's account. *localhost* solves this by using a plug-in called Celery. It provides a distributed task queue which delays the execution of certain jobs to a particular time ("Celery Project Documentation", 2018). In our use case, we set up periodic tasks for each property item so that it execute certain tasks when auction session is closed.

2.2 channels-redis

channels-redis is a Django Channels channel layer that uses Redis as its backing store ("channels-redis", 2018). This allows Django Channels to "communicate" across different ASGI instances, which is useful if we were to scale the website to run on a distributed system.

2.3 daphne

Daphne is a HTTP, HTTP2 and WebSocket protocol server for ASGI and ASGI-HTTP that was designed for Django Channels ("Daphne", 2018). It mainly handles WebSocket requests in our website, for example, bid requests, messaging and push notifications.

2.4 Django

The main framework this project runs on is Django. It is a high-level Python Web framework that allows the team to quickly prototype a working product with clean and pragmatic design. It is extremely scalable since it supports adding "apps" to a working site to extend its functionality ("Django", 2018).

2.5 Django Channels

Django Channels extends Django's ability beyond HTTP to handle WebSocket requests, chat protocols, etc. It allows communications between client and server using a long-running connection, while at the same time preserve

Django's synchronous and easy-to-use nature ("Django Channels Documentation", 2018). All of the real-time components of the project are implemented within the Django Channels framework, including auctions, messaging and push notifications.

2.6 django-celery-beat

This extension allows us to store Celery `PeriodicTask` in the database, which could then be modified during runtime ("Django Celery Beat Documentation", 2018). This is useful since we might have to change tasks execution interval if auction sessions are modified for a property item during run-time.

2.7 django-widget-tweaks

Widget-tweaks allows us to inject CSS attributes into form fields in templates ("Django Widget Tweaks", 2018). This means we can separate the business logic in "views" [1] and "templates". For example, if we want to add `class="form-control"` into a form input without widget-tweaks, we would need to append the class names in `__init__()` in the form class. With widget-tweaks, we can change the field representation in templates with Django template syntaxes (liquid tags).

2.8 Factory Boy

To generate our test data for both unit-tests and performance benchmark, we use Factory Boy to create random but reasonable test data. We uses an extension instead of creating fixtures on our own so that it is easier to maintain and the data is reproducible and consistent ("Factory Boy", 2018).

2.9 googlemaps

This is a Python Client for Google Maps Services which creates an abstraction between the Google Maps raw JSON API and our Django application ("Python Client for Google Maps", 2018). We uses this library to access the Maps API in backend.

2.10 gunicorn

Gunicorn is a Python WSGI HTTP Server for UNIX ("Gunicorn", 2018).

[1] Django view functions or Django class-based views

2.11 psycopg2-binary

psycopg2 is a binder/adapter between Python and PostgreSQL. It allows any Python application to connect to a PostgreSQL database instance ("Psycopg2", 2018).

2.12 redis (Python)

redis is a Python client for Redis ("Python Redis", 2018).

3 Licensing

Below is a list of license used by all the dependencies.

Dependency	Version	License
Automat	0.7.0	MIT
Django	2.1.2	BSD
Faker	0.9.1	MIT
Pillow	5.3.0	PIL
PyHamcrest	1.9.0	BSD
Twisted	18.7.0	MIT
aioredis	1.1.0	MIT
amqp	2.3.2	BSD
asgiref	2.3.2	BSD
async-timeout	3.0.0	Apache
attrs	18.2.0	MIT
autobahn	18.9.2	MIT
billiard	3.5.0.4	BSD
celery	4.1.1	BSD
certifi	2018.8.24	MPL-2.0
channels	2.1.3	BSD
channels-redis	2.3.0	BSD
chardet	3.0.4	LGPL
constantly	15.1.0	MIT
daphne	2.2.2	BSD
django-celery-beat	1.1.0	BSD
django-widget-tweaks	1.4.3	MIT
ephem	3.7.6.0	LGPL
factory-boy	2.11.1	MIT
googlemaps	3.0.2	Apache
gunicorn	19.9.0	MIT

hiredis	0.2.0	BSD
hyperlink	18.0.0	MIT
idna	2.7	BSD-like
incremental	17.5.0	MIT
kombu	4.2.1	BSD
msgpack	0.5.6	Apache
psycopg2-binary	2.7.5	LGPL
python-dateutil	2.7.3	Dual
pytz	2018.5	MIT
redis	2.10.6	MIT
requests	2.19.1	Apache
six	1.11.0	MIT
text-unidecode	1.2	Artistic
txaio	18.8.1	MIT
urllib3	1.23	MIT
vine	1.1.4	BSD
zope.interface	4.5.0	ZPL

4 Implementation

4.1 Architecture

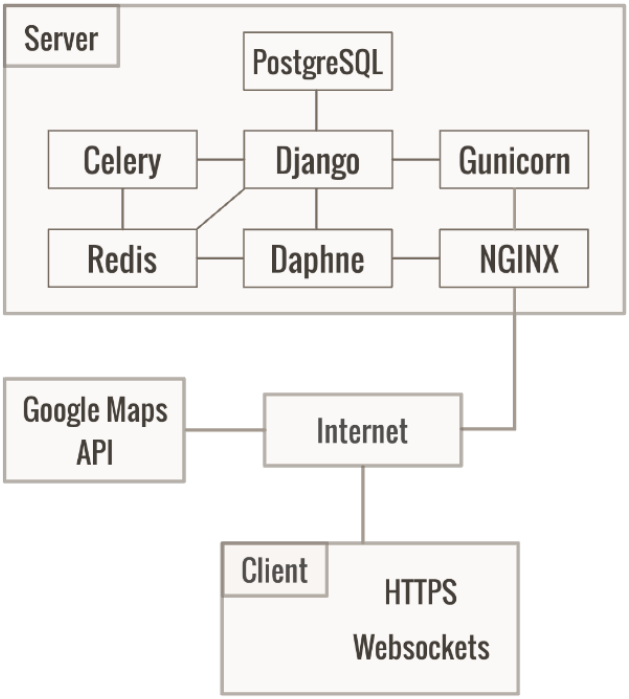

Figure 4.1.1: Architecture map for *localhost*

4.1.1 Execution Path

When navigating to *localhost*, the client is connected to the NGINX service. If the required page is a static file such as a CSS or JavaScript file then the file is immediately served by NGINX. WebSocket requests are forwarded to the Daphne service (see Section 4.4). All remaining requests are forwarded to the Gunicorn web server to then be redirected to Django.

4.1.2 Scalability

The architecture behind *localhost* was designed with scalability as a priority. The core services on the server are Django and PostgreSQL. Having multiple other services running concurrently reduces the computation time available for the core services. While *localhost* was deployed on a single server, the

architecture is designed so that services like NGINX can be deployed on seperate servers. As NGINX handles the majority of requests across the web, moving NGINX to a seperate server would significantly improve the performance of the platform. Each non-core service can have multiple workers and deployed on multiple servers, ensuring that *localhost* is scalable.

4.2 User Interface

Our vision for *localhost* was for a simple, modern and user friendly user interface across all devices as current solutions, whilst flexible, allow hosts to overload their listings which may cause them to lose potential tenants.

4.2.1 Cross browser and mobile compatibility

As we envisioned for *localhost* to operate on desktop and mobile devices, Bootstrap was deemed the most viable library to use as it is responsive and designed mobile-first. The challenge we faced using Bootstrap was that the custom templates for objects and designs were outdated and, while extremely functional and simple to prototype, was more difficult to restyle to suit our modern theme. This led us to discover Bootstrap Material, a fork of Bootstrap, which combined the responsiveness of Bootstrap with a modern and appealing interface.

4.2.2 Property Item Grouping

On most preexisting platforms, multiple listings in the same property or building would result in a separate search result returned for each listing. Our solution to this was to have a *Property* listing which would encapsulate all of the individual listings as *Property Items*. This is a major benefit as it simplifies the listing process for hosts, requiring them to upload only one set of pictures common to all the listings to the *Propety*, and unique images to each *Property Item*, hence, less storage space will be utilised on the server. Furthermore, this optimisation declutters the search results for prospective tenants, allowing them to view a larger variety of properties.

4.2.3 Information Limitation

To combat the information bloat that exists on most current services, our system enforces a word limit on both the description and title of each property and property item. Instead, we encourage hosts to post pictures of their property and select the amneties it has to offer which get reduced down to icons for consistency and clarity.

4.2.4 Google Maps API

The Google Maps API was used for this project because of its powerful search engine and its ability to scale with our architecture. The autocomplete on the search bars are powered by this and when a user selects a location to base their search on, the API breaks it down into the longitude and latitude values which get utilised in the back-end of the platform. Maps on each property page are also generated by the API.

4.2.5 Modifications from User Testing

We conducted a series of user tests, which can be viewed in the appendix, which were evaluated and factored into future revisions of the design of the user interface of *localhost*. A few of these included certain features being unclear or not obvious enough due to poorly coloured buttons or fonts, overly strict requirements on account creation, unoptimised process flow for users when performing certain actions such as not being able to message a host from their listing pages, and many more.

4.3 Database Design

4.3.1 Database of localhost

An ER diagram of our database can be found in Appendix. The database of *localhost* is normalised in 3NF (third normal form) meaning it contains no transitive dependencies, minimising our data redundancy and improving data integrity. Extensibility was kept in mind when designing the database schema.

4.3.2 Property Search Time Optimisation

One of the significant features in the application is searching for properties based on their distance. User can enter an address in the search bar and properties near the search location will be displayed in the results page, sorted by distance. There are several ways to implement the property search and we considered three implementations:

1. Uses GeoDjango with PostGIS, and migrate our property model to use geometry field in GeoDjango

2. stores latitude and longitude for each property. Then use an extension "Geopy" to approximate distances between properties in `QuerySet` and the search location ("GeoPy Documentation", 2018)

3. stores latitude and longitude for each property. Then annotate each item in `QuerySet` with built-in SQL function expressions to calculate the distance between search location and each property, then sort by that distance

Implementation 1 provides the most accurate distance approximations as GeoDjango supports querying and manipulating spatial data in database, which takes into account elevations and sea level ("GeoDjango Raster Lookups", 2018). However, the accuracy gained from this framework is negligible when compared to the accuracy of Implementations 2 and 3. Our final implementation was accurate to within 50m of the target location, which is enough accuracy for sorting properties by distance. The effort to set up and migrate to a new database framework was thought to be too much work for little gain.

Implementation 2 which calculates distances using Geopy was our second consideration. Algorithm 1 roughly shows how we implemented the search and sorting using GeoPy.

Algorithm 1 Inefficient implementation of property search

$qs \leftarrow Property.filter(critieria)$
$lat, lng \leftarrow urlparam.get(lat), urlparam.get(lng)$
$properties \leftarrow list()$
for all *property* in qs **do**
  $distance \leftarrow geopy.distance((lat, lng), (property.lat, property.lng))$
  $properties.append(property, distance)$
end for
$properties \leftarrow sorted(properties, key = \lambda x : x[1])$

Algorithm 1 works fine but the performance drops as number of properties grows. Distances are calculated using Python calls which could be far slower than SQL functions, since most of the computations are done using GeoPy. For a large dataset of properties, performance would be poor.

Having considered these alternatives, we chose to use Implementation 3 which used native SQL expressions to compute the distances which would greatly improve performance. To approximate distances using native SQL expressions, we need a function/formula to translate latitudes and longitudes in the database model to actual distances on the Earth.

Haversine Formula The haversine formula is used to determine the great-circle distance between two points on a sphere given their longitudes and

latitudes (de Mendoza y Rios, 1796).

$$d = 2r \arcsin \sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}$$

where,

- ϕ_1, ϕ_2 are latitude of search location and latitude of property
- λ_1, λ_2 are longitude of search location and longitude of property.
- r is the radius of the Earth

Our search query calculates the distance from each property to the target destination using Haversine's Formula and then orders them by closest distance. Since it is costly to calculate this for every property in the database, an initial filter was added to narrow down the number of properties that the distance calculation had to be operated on.

Only properties within ± 0.15 latitude and longitude offsets of the original search, which translates to around 15km have their distance approximated using Haversine's Formula. This greatly improves the search times and also removes properties that are outside a reasonable distance from the search. By adding the initial filter, our search time is also no longer restricted by the property dataset and is instead limited by suburb density.

As can be seen in Figure 4.3.1, the search time for large property data sets appears to increase linearly. For a database with one million properties, it takes only 0.04 seconds to complete the search. This can be mostly be attributed to the initial filter, which drastically cuts down search time.

4.4 Real Time Communication

The real time communication powering platform features such as notifications, instant messaging, and bidding events, is provided through the use of web sockets. WebSockets allow for bi-directional messages to be instantly communicated with minimal overhead ("What are WebSockets?", 2018). While Django doesn't provide support for WebSockets, support can be provided through a third-party extension for Django called Django Channels.

4.4.1 MSocket Library & Django Channels

Django Channels does not provide native support for multiplex sockets. When a client application begins listening to an event it must dedicate an

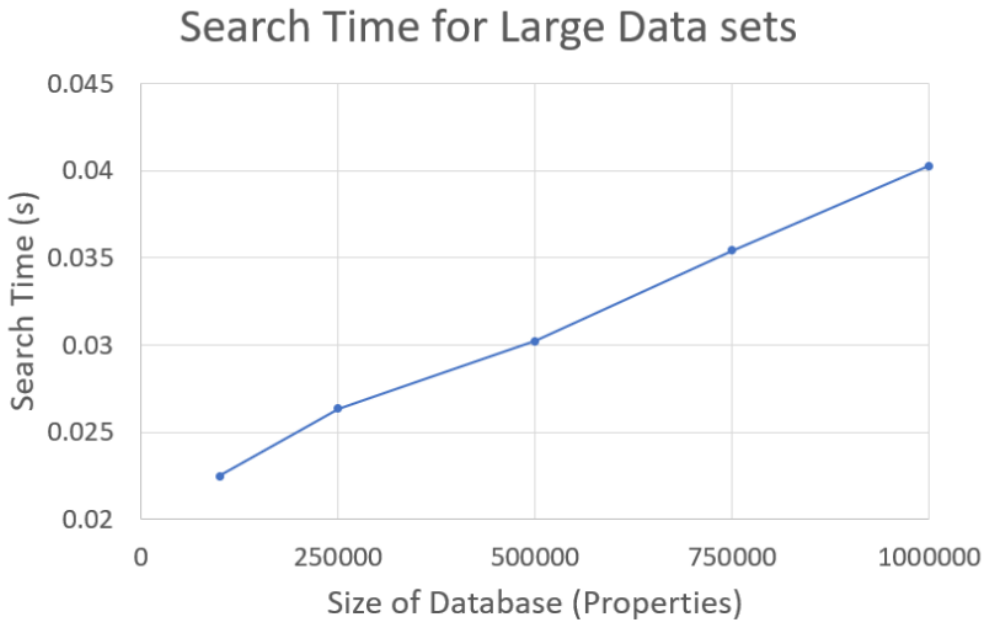

Figure 4.3.1: Performance of large data sets

entire socket. Consequently, a client may have multiple sockets open to send and recieve messages for individual events despite the fact that the traffic could be sent over a single socket. A server only has a limited number of sockets available so this reduces the scalability by a significant factor. Multiplex sockets are sockets that are used to transmit information of different classes that are then filtered by the reciever and forwarded to the correct recipient (See Figure 4.4.1).

To ensure that *localhost* was scalable, support for multiplex sockets was provided with a custom client-side JavaScript library and custom Django class. The client-side library, called *msocket.js*, is a cooperative multiplex socket libary. This means that for the socket multiplexing to work, the client and server must adopt a common protocol of communication. The protocol used in *localhost* is provided below in the form of a JSON file.

```
{
  type: ,
  data: {

  }
}
```

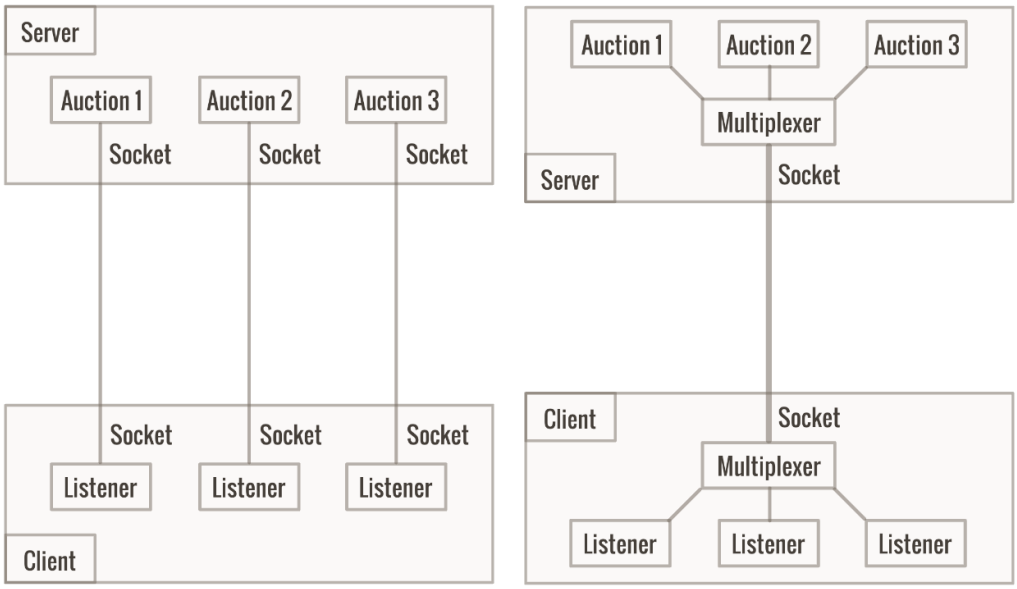

Figure 4.4.1: Comparison between regular sockets and multiplexed sockets

The type allows the multiplexer to determine how to handle the message, and the data is the payload for the message. In the case of real-time bidding, the payload would contain details allowing the handler to determine which property item the bid is for, and how much the bid is.

msocket.js is designed from scratch to provide a simple and abstracted interface to hide away the details and complexity behind WebSockets. A client can submit handlers to execute for different types (shown below) and they will automatically execute as messages of that type arrive.

```
/**
 * @brief Registers a handler for a message type
 *
 * @param type  The type of message the handler should apply to
 * @param handler The handler to execute on message recieve
 *              Should take the data as an argument
 *              See JSON specification
 *
 * @return On success, @c true
 * @return On failure, @c false when a handler is already set for
    the type
 * */
register_handler(type, handler);
```

Server-side, the Django Channels socket consumer class was modified

to maintain a record of each of the events the socket is subscribed to and whenever a message is broadcasted for one of these events the consumer transmits it to the client with the appropriate class identifier.

4.5 Event Scheduling

When an auction session ends, tasks need to be performed in order to complete the bidding process. There are also several places where scheduled/delayed tasks are needed in the application.

Property items are said to be "available" if the *available* option is turned on by user (it is "on" by default when created) and there are no bids in all the previous auction sessions in the same day. When the local time reaches the end time of the session, the system have to check whether there are bids in the session and create a booking for the winner. Property items that has been booked out has to be marked as "unavailable" so that it does not come up in the search results. Furthermore, property item has to be re-listed (marked as "available") at 12 noon every day.

4.5.1 Django Signals

Signals are dispatched in Django when a certain action is performed within the framework. It helps decoupled applications to get notified when the actions are taken placed ("Django Signals Documentation", 2018). Django provides a set of built-in signals that allows us to combine it with Celery to perform certain tasks when signals are dispatched. A list of signals we used in the project are:

- `django.db.models.signals.m2m_changed`
- `django.db.models.signals.pre_save`

4.5.2 Celery Beat

Celery beat is a scheduler in Celery that kicks off tasks at regular intervals, that are then executed by available worker nodes in the cluster ("Celery Beat Documentation", 2018). The worker works asynchronously to start task execution and store results in a Redis instance.

By default, beat uses `PersistentScheduler` that keeps track of the last run in a local shelve[2] database file. But that is not necessary since we already have a PostgreSQL database active in the back-end. So we instead use an

[2]a persistent, dictionary-like object

extension django-celery-beat 2.6 that allows us to store tasks in the Django database through a `DatabaseScheduler`. The extension also provides an admin interface so that tasks could be managed by the administrator.

Periodic Task `PeriodicTask` is a Django model provided by django-celery-beat 2.6 that simulates how Celery tasks are represented in the database. We use this in conjunction with `DatabaseScheduler` to provide background services in the Django framework.

4.5.3 Combining both signals and Celery beat

There are two tasks to be completed in the background:

1. check if anyone bid on a property item; if there is, a booking is added to the winner's account, and bids associated with the property item are removed. Property item is marked as "unavailable" as well. Otherwise we do nothing about that property item.

2. enable bidding for all property items at 12 noon

As such, we define two tasks in Celery accordingly: `cleanup_bids` and `enable_bids`. Whenever an auction session is added to a property item (`m2m_changed`), we add a `PeriodicTask` to the database that executes `cleanup_bids`. A `PeriodicTask` that executes `enable_bids` is added when a property item is saved (`pre_save`). When celery is launched, the scheduler will pick up the tasks and triggers tasks execution when time reaches.

5 Design

5.1 Architecture

5.2 User Interface

5.3 Database

5.3.1 PostgreSQL

PostgreSQL was used for this project because of its high scability and stability. Complex queries are performed faster in PostgreSQL than alternatives such as MySQL which is important for optimising our search times especially for a large property dataset. PostgreSQL is also ACID (Atomicity, Consistency, Isolation, Durability) compliant, which ensures that no data is lost in the system in the case of failures. Since we receive a large number of database read and writes towards the end of an auction session, concurrency control is vital to ensure the consistency of data. PostgreSQL efficiently handles concurrency with its MVCC (Multiversion Concurrency Control). Each database query sees only a snapshot of the database when it was accessed, preventing the query from seeing inconsistent data that could be caused by other concurrent updates on the same data. PostgreSQL is also horizontally scalable as it supports data replication and sharding. This is important if our project was to be extended and deployed outside the NSW region and potentially globally.

5.3.2 Database of localhost

A full entity relationship diagram of our database can be found in Appendix. The database of *localhost* is normalised in 3NF (third normal form) as it contains no transitive dependencies, minimising our data redundancy and improving data integrity. Extensibility was kept in mind when designing the database schema.

5.3.3 Search Time Optimisation

Property locations are determined by their latitude and longitude stored in the database. When a user searches a location, the Google Maps API converts the location into a latitude and longitude which is then used as the target destination. The results of our property search are shown in order of closest distance to the target destination. Distance calculation is performed using Haversine's formula, which determines the great-circle distance

between two points in NSW. Vincenty's formula is a popular and more accurate alternative for calculating distance, however we chose to use Haversine's formula as it is computationally much faster.

Listing 5.3.1: Haversine's Formula

```
radlat = Radians(latitude)                  # Destination Lat
radlong = Radians(longitude)                # Destination Long
radflat = Radians(models.F('latitude'))     # Property Lat
radflong = Radians(models.F('longitude'))   # Property Long

distance = 6371 * Acos(
    Cos(radlat) * Cos(radflat) * Cos(radflong - radlong) +
    Sin(radlat) * Sin(radflat))
```

Our search query calculates the distance from each property to the target destination using Haversine's Formula and then orders them by closest distance. Since it is costly to calculate this for every property in the database, an initial filter was added to narrow the number of properties that the distance calculation had to be operated on.

Listing 5.3.2: Initial filter to narrow search results

```
lat_offset, lng_offset = Decimal(0.15), Decimal(0.15)
lat_range = (lat - lat_offset, lat + lat_offset)
lng_range = (lng - lng_offset, lng + lng_offset)
properties = Property.objects.within(lat, lng).filter(
    latitude__range=lat_range, longitude__range=lng_range)
```

Only properties within ± 0.15 latitude and longitude offsets of the original search, which translates to around 15km, are shown. This greatly improves the search times as well as removes properties that were outside a reasonable distance from the search. By adding the initial filter, our search time is also no longer restricted by the property dataset and is instead limited by suburb density.

5.4 Event Scheduling

5.5 Real-Time Communication

6 Manual

6.1 Deployment

localhost is deployed using Amazon Web Services on an Amazon Elastic Compute Cloud instance with the following specifications.

```
OS: Ubuntu 18.04.1 LTS x86_64
Host: HVM domU 4.2.amazon
Kernel: 4.15.0-1023-aws
CPU: Intel Xeon E5-2676 v3 (1) @ 2.4
GPU: Cirrus Logic GD 5446
Memory: 983MiB
```

6.1.1 Amazon Web Services Configuration

After creating an Amazon Elastic Compute Cloud instance through the Amazon Web Services dashboard, the only port on the server accessible from the internet is `22 (SSH)`. For the server to accept web requests, ports `80 (HTTP)` and `443 (SSL)` must be opened. This can be done by navigating to the dashboard, selecting the running *Amazon Elastic Compute Cloud* instance, and selecting `launch-wizard`. A tab will then appear at the bottom of the page and will contain a form that can open ports.

6.1.2 Software

While a number of scripts are provided to automate the installation and configuration of *localhost*, some manual configuration is required. The configuration required varies between different Linux distribution so the following steps are only supported on Ubuntu 18.04.1.

The default user account provided on an Amazon Elastic Compute Cloud instance running Ubuntu 18.04.1 is `ubuntu` and the following commands rely on this assumption. Commands requiring sudo privileges are prepended with `#` and commands executed under user permissions are prepended with `$`.

Update The initial state of the instance is likely out of date and should be brought up to date by running the following command.

```
# apt update && apt upgrade && apt dist-upgrade && apt autoremove
```

Repository To ease updating, an SSH key will be generated to authenticate the server when accessing the repository.

```
$ ssh-keygen -t rsa -b 4096
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in id_rsa.
Your public key has been saved in id_rsa.pub.
```

After adding `/home/ubuntu/.ssh/id_rsa.pub` to BitBucket the repository can be cloned.

```
$ git clone git@bitbucket.org:jtalowell/localhost.git
   /home/ubuntu/localhost
```

Django & Immediate Dependencies To install a virtual environment and manage dependencies the following packages must be installed.

```
# apt install python3-venv python3-pip
```

Django and its immediate dependencies can then be installed.

```
$ cd /home/ubuntu/localhost
$ python3 -m venv venv
$ source venv/bin/activate
$ (venv) pip install -r requirements.txt
```

PostgreSQL PostgreSQL must be installed and it's accompaning system service enabled.

```
# apt install postgresql postgresql-contrib
# systemctl enable postgresql.service
# systemctl start postgresql.service
```

The database cluster can then be created.

```
$ sudo -u postgres -i
[postgres]$ /usr/lib/postgresql/10/bin/initdb -D
   '/var/lib/postgresql/data'
```

A user to administer the database clusted must be created for access.

```
[postgres]$ createuser --interactive
Enter name of role to add: ubuntu
```

```
Shall the new role be a superuser? (y/n) y
[postgres]$ exit
```

The database to use for the project can now be created.

```
$ createdb localhost_db
$ sudo -u postgres -i
[postgres]$ psql
postgres=# \password ubuntu
Enter new password: [DB_PW]
Enter it again: [DB_PW]
postgres=# GRANT ALL PRIVILEGES ON DATABASE localhost_db TO ubuntu;
postgres=# ALTER ROLE ubuntu SET client_encoding TO ‘‘utf8’’;
postgres=# ALTER ROLE ubuntu SET default_transaction_isolation TO
    ‘‘read committed’’;
postgres=# \q
[postgres]$ exit
```

Redis Ubuntu's package for Redis works without any extended configuration and its services are automatically started and enabled.

```
# apt install redis-server
```

Daphne Daphne is installed as one of Django's immediate dependencies. A service script is provided for installation.

```
# ln -sf /home/ubuntu/localhost/deploy/systemd/daphne.service
    /etc/systemd/system/
# vim /etc/systemd/system/daphne.service
```

The environment variables for `SECRET_KEY`, `DB_USER`, and `DB_PW` must be set for Daphne to be able to operate. This can be done by inserting the following lines in the service script.

```
Environment="SECRET_KEY=YOUR KEY HERE"
Environment="DB_USER=ubuntu"
Environment="DB_PW=YOUR DB PASSWORD HERE"
```

Gunicorn Gunicorn, like Daphne, is installed as one of Django's immediate dependencies. A service script is provided for installation.

```
# ln -sf /home/ubuntu/localhost/deploy/systemd/gunicorn.service
    /etc/systemd/system/
```

```
# vim /etc/systemd/system/gunicorn.service
```

The environment variables for `SECRET_KEY`, `DB_USER`, and `DB_PW` must be set for Gunicorn to be able to operate. This can be done by inserting the following lines in the service script.

```
Environment="SECRET_KEY=YOUR KEY HERE"
Environment="DB_USER=ubuntu"
Environment="DB_PW=YOUR DB PASSWORD HERE"
```

Celery Service scripts are provied for both Celery and Celery Beat.

```
# ln -sf /home/ubuntu/localhost/deploy/systemd/celery.service
   /etc/systemd/system/
# ln -sf /home/ubuntu/localhost/deploy/systemd/celery-beat.service
   /etc/systemd/system/
# vim /etc/systemd/system/celery.service
# vim /etc/systemd/system/celery-beat.service
```

The following must be added to both service scripts.

```
Environment="SECRET_KEY=YOUR KEY HERE"
Environment="DB_USER=ubuntu"
Environment="DB_PW=YOUR DB PASSWORD HERE"
```

NGINX NGINX can be installed using the following commands.

```
# apt install nginx
# systemd enable nginx
# systemd start nginx
```

The relevant configuration files can be installed by:

```
# /home/ubuntu/localhost/deploy/init.sh
```

localhost uses Let's Encrypt as the provider for it's SSL certificates. The certificates can be installed using the Certbot tool which can be installed and executed by the following commands. For this to work, the domain name registered for *localhost* should direct to the IP address of the Amazon Elastic Compute Cloud instance.

```
# apt install software-properties-common
# add-apt-repository ppa:certbot/certbot
# apt update
```

```
# apt install python-certbot-nginx
# certbot --nginx certonly
```

After installing the certificate the `init.sh` script should be executed again.

```
# /home/ubuntu/localhost/deploy/init.sh [SECRET_KEY] [DB_USER]
    [DB_PW]
```

This script first ensures that the NGINX configuration script is installed and restarts all the services that *localhost* depends on.

Maintenance There are two scripts that are provided to ease server upgrades and deployment testing. The first script is `rebuild.sh`. This script automates the dropping, creation, migrations, and loading of test data for the database. This is useful for quickly loading a usable set of test data without manually creating listings and accounts. It's also extendable to restore backups.

```
-M is short for --Makemigrations
-m is short for --migrate
-s is short for --server and should only be used on the server
    deployment
-l is short for --loaddata and loads both testdata and
    propertyimages
```

The second script has already been mentioned but remains relevent for future updates. `init.sh` should be ran every time any updates or upgrades take place to restart the required services otherwise unexpected errors can occur through caching. When making changes to scheduling times, the Celery and Celery Beat services must be stopped prior to running the `init.sh` script.

```
# /home/ubuntu/localhost/deploy/init.sh
```

References

Celery Beat Documentation. (2018). Retrieved from http://docs.celeryproject.org/en/latest/userguide/periodic-tasks.html

Celery Project Documentation. (2018). Retrieved from http://docs.celeryproject.org/en/latest/index.html

channels-redis. (2018). Retrieved from https://github.com/django/channels_redis

Daphne. (2018). Retrieved from https://github.com/django/daphne/

de Mendoza y Rios, D. J. (1796). *F.r.s. recherches sur les principaux problemes de l'astronomie nautique.* (Vol. 87). Philosophical Transactions of the Royal Society of London.

Django. (2018). Retrieved from https://www.djangoproject.com/

Django Celery Beat Documentation. (2018). Retrieved from https://django-celery-beat.readthedocs.io/en/latest/

Django Channels Documentation. (2018). Retrieved from https://channels.readthedocs.io/en/latest/

Django Signals Documentation. (2018). Retrieved from https://docs.djangoproject.com/en/2.1/topics/signals/

Django Widget Tweaks. (2018). Retrieved from https://github.com/jazzband/django-widget-tweaks

Factory Boy. (2018). Retrieved from https://factoryboy.readthedocs.io/en/latest/

GeoDjango Raster Lookups. (2018). Retrieved from https://docs.djangoproject.com/en/2.1/ref/contrib/gis/db-api/#raster-lookups

GeoPy Documentation. (2018). Retrieved from https://geopy.readthedocs.io/en/stable/

Gunicorn. (2018). Retrieved from https://gunicorn.org/

Psycopg2. (2018). Retrieved from http://initd.org/psycopg/

Python Client for Google Maps. (2018). Retrieved from https://github.com/googlemaps/google-maps-services-python

Python Redis. (2018). Retrieved from https://redislabs.com/lp/python-redis/

What are WebSockets? (2018). Retrieved from https://pusher.com/websockets

A User Tests

A.1 Andy

```
###### Version 0.1.0 - 06 Oct, 2018
###### Supervisor: Edward
###### Tester: Andy
###### Completion Date: 06 Oct
###### Completion Time (24HR):

# Instruction Set
1. **Create a user account and login**
*User difficulty rating (Out of 5):* 3
*Supervisor comments*:
Too many security requirements for password
Should be a pop-up or modal to prompt user that account has been
    created. Any confirmation.
Recommends that dob and gender should not be required for signup,
    but part of profile creation
that user can do later. Lower barrier of entry

2. **Add $150 to your wallet**
*User difficulty rating (Out of 5):* 4 for navigation
*Supervisor comments*:
Tabs are faint on dashboard, hard for people with poor eyesight to
    see
Confirmed for recharge success

3. **Search for a listing in kingsford**
*User difficulty rating (Out of 5):* 3, could be higher if
    navigation was clearer
*Supervisor comments*:
Needs a clearer home button on dashboard
The search bar was not distinguishable
Since enter button doesn't work, instead of saying "Try
    Kingsford", tell them to select Kingsford or something

4. **Submit a bid on first listing in first property in search
    results**
*User difficulty rating (Out of 5):* 3
*Supervisor comments*:
Confirmation that bid was succesful
Listing requires more information
```

```
An actual web-page back button, rather than just the browser back
Should be clearer that you're the current top bidder
More description needed for the rooms, detail lacking

5. **Find the list of user's currently active bids**
*User difficulty rating (Out of 5):* 4
*Supervisor comments*:
Activity tab name is bit vague
Countdown timer for how long until auction ends
Needs more information: property name, address
Maybe potential to re-bid on activity page

6. **Navigate to the listing that was previously bid on without
   searching the location**
*user difficulty rating (out of 5):* 1
*supervisor comments*:
Easy to navigate, but activity page was lacking description
Refer to comments in 5

7. **Message owner of the listing previously bid on**
*user difficulty rating (out of 5):* 1
*supervisor comments*:
Button to directly message host, instead of going to host profile
Message should refer to the property that was in question

8. **Create a listing**
*User difficulty rating (Out of 5):* 2
*Supervisor comments*:
Amenities list is slightly unintuitive. Bubbles?
Headings need to be reworded. New Item needs to be renamed

9. **Navigate to the home page**
*User difficulty rating (Out of 5):* 2
*Supervisor comments*:

10. **Find the listing that was created without searching the
   location**
*User difficulty rating (Out of 5):* 4
*Supervisor comments*:

11. **Edit user profile**
*user difficulty rating (out of 5):* 2
*supervisor comments*:
```

```
Unclear what you are editing

12. **View user profile**
*user difficulty rating (out of 5):* 4
*supervisor comments*:

# Overall Thoughts
*Supervisor comments*:
*User comments*:
First glance, the home page is unclear what the purpose of the
    site is
```

A.2 Anish

```
##### Version 0.1.0 - 06 Oct, 2018
###### Supervisor: Ed
###### Tester: Anish
###### Completion Date: 6 Oct
###### Completion Time (24HR):

# Instruction Set
1. **Create a user account and login**
*User difficulty rating (Out of 5):* 3
*Supervisor comments*:
Password confirmation unclear, impressed by password security

2. **Add $150 to your wallet**
*User difficulty rating (Out of 5):* 3
*Supervisor comments*:
Overflow when he tried to add 1000000000000

3. **Search for a listing in Kensington**
*User difficulty rating (Out of 5):* 2
*Supervisor comments*:
From dashboard, hard to get to search.
Search bar wasn't clear enough

4. **Submit a bid on first listing in first property in search
    results**
*User difficulty rating (Out of 5):* 3
*Supervisor comments*:
Doesn't show user has winning bid
```

```
Should have link to wallet if not enough funds

5. **Find the list of user's currently active bids**
*User difficulty rating (Out of 5):* 4
*Supervisor comments*:

6. **Navigate to the listing that was previously bid on without
    searching the location**
*user difficulty rating (out of 5):* 5
*supervisor comments*:

7. **Message owner of the listing previously bid on**
*user difficulty rating (out of 5):* 5
*supervisor comments*:

8. **Create a listing**
*User difficulty rating (Out of 5):* 2
*Supervisor comments*:
Button was hard to find

9. **Navigate to the home page**
*User difficulty rating (Out of 5):* 2
*Supervisor comments*:

10. **Find the listing that was created without searching the
    location**
*User difficulty rating (Out of 5):* 5
*Supervisor comments*:

11. **Edit user profile**
*user difficulty rating (out of 5):* 4
*supervisor comments*:

12. **View user profile**
*user difficulty rating (out of 5):* 5
*supervisor comments*:

# Overall Thoughts
*Supervisor comments*:
*User comments*:
overal pretty intuitive, however some of the buttons were pretty
    tricky to find, (e.g. home pages, adding property), and the
    fonts could generally be bigger
```

A.3 Ming

```
###### Version 0.1.0 - 06 Oct, 2018
###### Supervisor: Edward
###### Tester: Ming
###### Completion Date: 6 Oct
###### Completion Time (24HR):

# Instruction Set
1. **Create a user account and login**
*User difficulty rating (Out of 5):* 3
*Supervisor comments*:
Wanted to register as 'other' gender
Maybe age restriction on dob

2. **Add $150 to your wallet**
*User difficulty rating (Out of 5):* 3
*Supervisor comments*:
Call dashboard 'account' or something more intuitive
Confirmation before and after money is added

3. **Search for a listing in kingsford**
*User difficulty rating (Out of 5):* 2
*Supervisor comments*:
Search bar was hard to see in background
Had to navigate to home before search

4. **Submit a bid on first listing in first property in search
   results**
*User difficulty rating (Out of 5):* 3
*Supervisor comments*:

5. **Find the list of user's currently active bids**
*User difficulty rating (Out of 5):* 1
*Supervisor comments*:
Looked inside booking history
Bids should be under a subcategory in Activity, or rename Activity

6. **Navigate to the listing that was previously bid on without
   searching the location**
*user difficulty rating (out of 5):* 4
*supervisor comments*:
```

```
7. **Message owner of the listing previously bid on**
*user difficulty rating (out of 5):* 2
*supervisor comments*:
Should be a button that can directly message, rather than go to
   profile
Should link property/listing in question to the message

8. **Create a listing**
*User difficulty rating (Out of 5):* 2
*Supervisor comments*:
Button out of place

9. **Navigate to the home page**
*User difficulty rating (Out of 5):* 2
*Supervisor comments*:
Should be top left, on dashboard page

10. **Find the listing that was created without searching the
   location**
*User difficulty rating (Out of 5):* 4
*Supervisor comments*:

11. **Edit user profile**
*user difficulty rating (out of 5):*
*supervisor comments*: 2
Form should be clearer, should say changing description.
Didn't know if he was changing dob, name, etc

12. **View user profile**
*user difficulty rating (out of 5):*
*supervisor comments*: 4

# Overall Thoughts
*Supervisor comments*:
*User comments*: 2.5/5
Hard to navigate, things are still unpolished and unclear. Make
   things more obvious
Eg. Home button should consistently be at top left, adding
   property button, search bar is hard to see
More specific titles
```

A.4 Weilon

```
###### Version 0.1.0 - 06 Oct, 2018
###### Supervisor: Thomas Lam
###### Tester: Weilon Ying
###### Completion Date: 06-10-2018
###### Completion Time (24HR): 14:00:00

# Instruction Set
1. **Create a user account and login**
*User difficulty rating (Out of 5):* 4
*Supervisor comments*:
Didn't know that the date field is for date of birth

2. **Add $150 to your wallet**
*User difficulty rating (Out of 5):* 5
*Supervisor comments*:

3. **Search for a listing in kingsford**
*User difficulty rating (Out of 5):* 5
*Supervisor comments*:
Search bar is hard to see. Recommendation by user: make background
   darker.

4. **Submit a bid on first listing in first property in search
   results**
*User difficulty rating (Out of 5):* 3
*Supervisor comments*:

5. **Find the list of user's currently active bids**
*User difficulty rating (Out of 5):* 5
*Supervisor comments*:

6. **Navigate to the listing that was previously bid on without
   searching the location**
*user difficulty rating (out of 5):* 5
*supervisor comments*:

7. **Message owner of the listing previously bid on**
*user difficulty rating (out of 5):* N/A
*supervisor comments*:

8. **Create a listing**
*User difficulty rating (Out of 5):* 4
*Supervisor comments*:
```

```
Cannot remove property item if accidentally press +add button.

9. **Navigate to the home page**
*User difficulty rating (Out of 5):* 5
*Supervisor comments*:

10. **Find the listing that was created without searching the
    location**
*User difficulty rating (Out of 5):* 5
*Supervisor comments*:

11. **Edit user profile**
*user difficulty rating (out of 5):* 5
*supervisor comments*:

12. **View user profile**
*user difficulty rating (out of 5):* 5
*supervisor comments*:

# Overall Thoughts
*Supervisor comments*:
User likes the minimalistic design.

*User comments*:
User suggested to propagate ?next from login to registration.
```