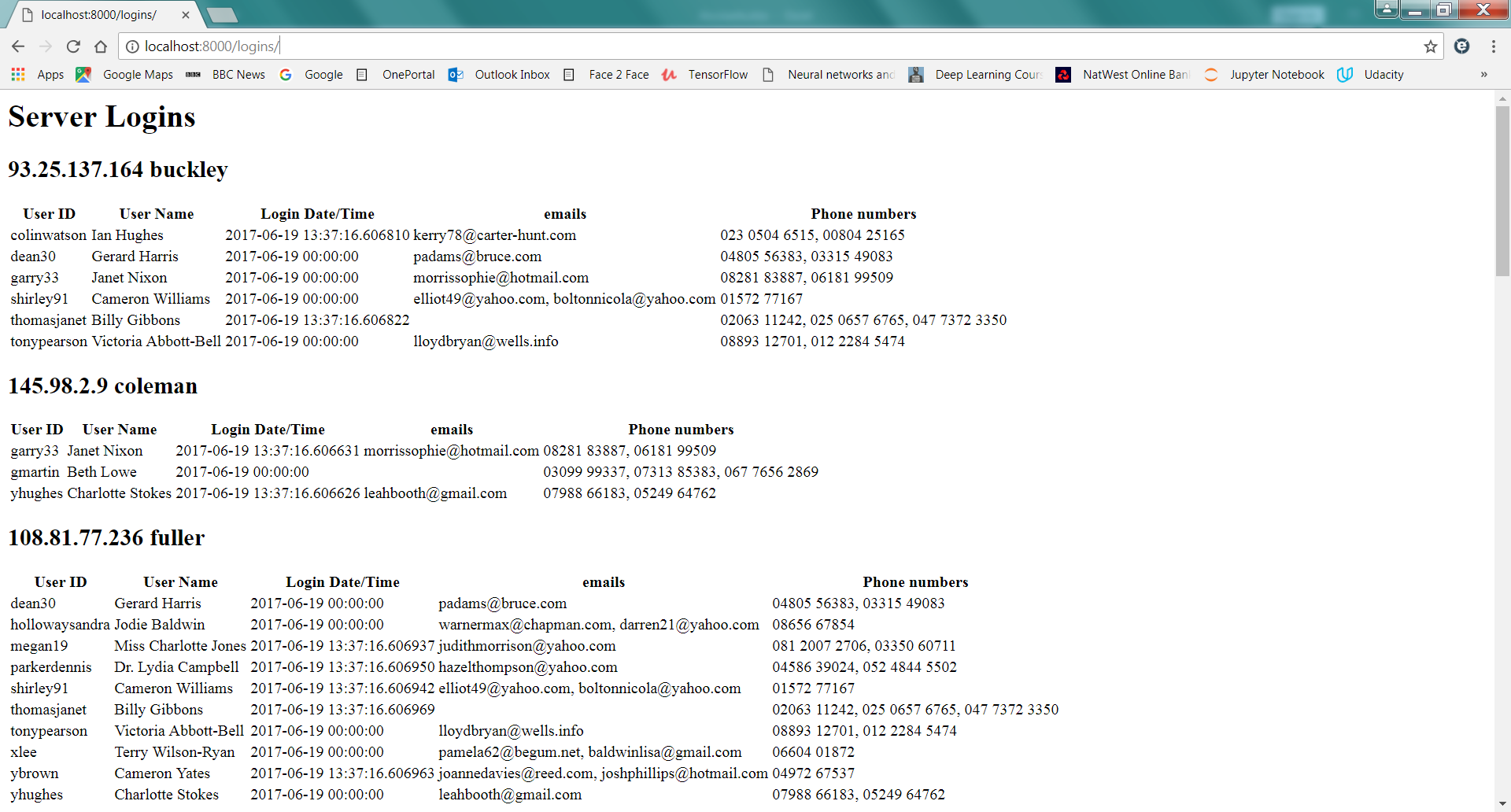
**Notes on Mid-Level Interview project**

Screenshot of output:



**Database Population**

I created the models in models.py by reverse-engineering them from tables I had already created in sqlite using inspectdb. This was a bit problematic due to Django’s need for ID fields unless there is a single primary key – a problem on link tables. The implementation gets around this by assigning primary key references to the defined foreign keys – I’m actually surprised that this works. With more time, I’d redo this so that the models form the starting point for the schema.

I renamed import.py to import\_csv.py as “import” as a module name was upsetting my IDE (PyCharm). The import\_csv routine was written first and uses direct database writes rather than integrating the models that I later derived from the schema. This would be an area to tidy up later.

The logic for IPs, emails, phone numbers and datetimes is on a best efforts basis. For example, to keep things simple I assumed that any phone numbers using the +44 prefix would also include the leading zero in some form (this is actually not how the convention should work). The phone numbers are also assumed to be UK numbers. I grouped them 5/5 for 10-digit numbers and 3/4/4 for 11-digit numbers, but the true formatting rules for 11-digit numbers are much more complex. For dubious dates I considered all possible permutations and chose the resulting valid date (not in the future) that was nearest to “now”, on the basis that when processing a real log of server login attempts, the datetimes in the log would almost certainly be quite recent.

**Database Queries**

The tables containing users, servers, user emails and user phone numbers are required fully once at the start of the view. These tables are always likely to be small and so no appreciable overhead is involved, on the assumption that the view is executed relatively infrequently (if it were queried say multiple times per second by many different users then there would be an argument for periodic cached queries to get this relatively static information).

A query is then made for each user/server combination to return the latest login for that combination. I imagine that this would translate into a “top 1” style database query so this should be fast even when the server logins table becomes very large, when using a single query to return all logins would be prohibitive.