Team 11 Final Report

Alchan Kim, Jiho Yu, and Kyunggeun Lee

1. Abstract

KiWi Order is a flexible automatic ordering system for food stores such as coffee shops or restaurants. By integrating automatic kiosk and remote ordering service(e.g. using smart phones), *KiWi Order* will provide store managers with an easier, accessible solution for managing both customers who order at the store and those who order with their smart phones.

2. Motivation

Considering recent tendencies in Korean economy, kiosk has been expected by many to be an attractive option for many businesses. However, the market of kiosk is currently not as activated as the expectation because they have problems of being costly and inflexible. For example, as a shop manager, employing a kiosk is expensive, while they are not sure about the positive effect of it. For customers, kiosk sometimes causes unnecessary inconvenience such as long lines, unfamiliar interface, etc. Therefore, our project aims to develop a more cost-efficient and flexible solution for both managers and customers.

3. Related Work

CI Tech is the manufacturer of the currently dominant unmanned kiosk products for food stores. Therefore, we set the service they offer as baseline. *CI Tech* offers kiosk software including kiosk machine itself, at approximately $3,000 per machine. However, we doubt that this is optimal. If we can detach the kiosk service from hardware, we believe that we can 1. cut down price dramatically, and 2. enhance flexibility of kiosk service.

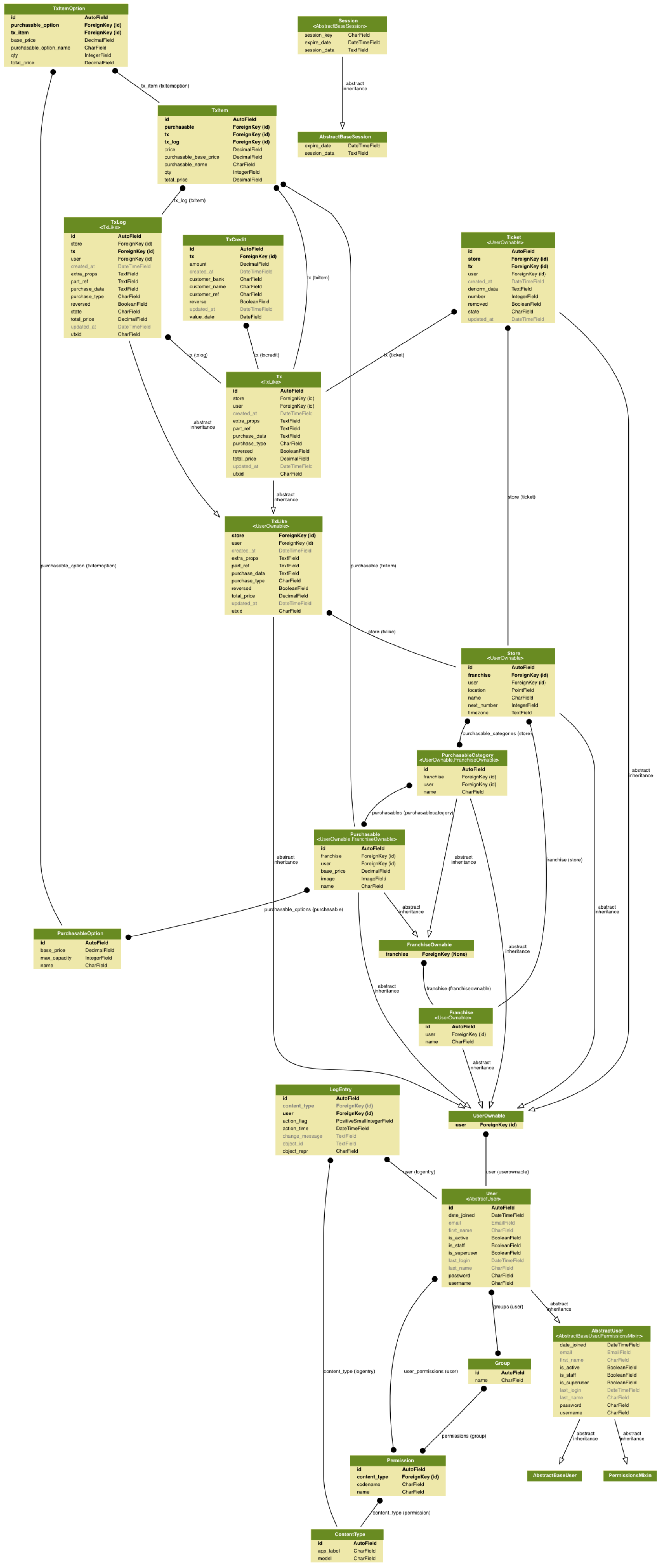
Being a SaaS, *KiWi Order* is machine-independent; instead of machines which cost about $3,000 each, we believe that an iPad would be decent enough for servicing *KiWi Order*, costing only about $500(latest model of iPad 9.7). Moreover, the software updates will be 100% free, since *KiWi Order* is SaaS. Unlike the existing kiosk systems which only provides kiosk service, *KiWi Order* also provides remote ordering service in a single system. With *KiWi Order*, managers will no more have to worry about integrating the two.

4. Design

4.1. System Architecture

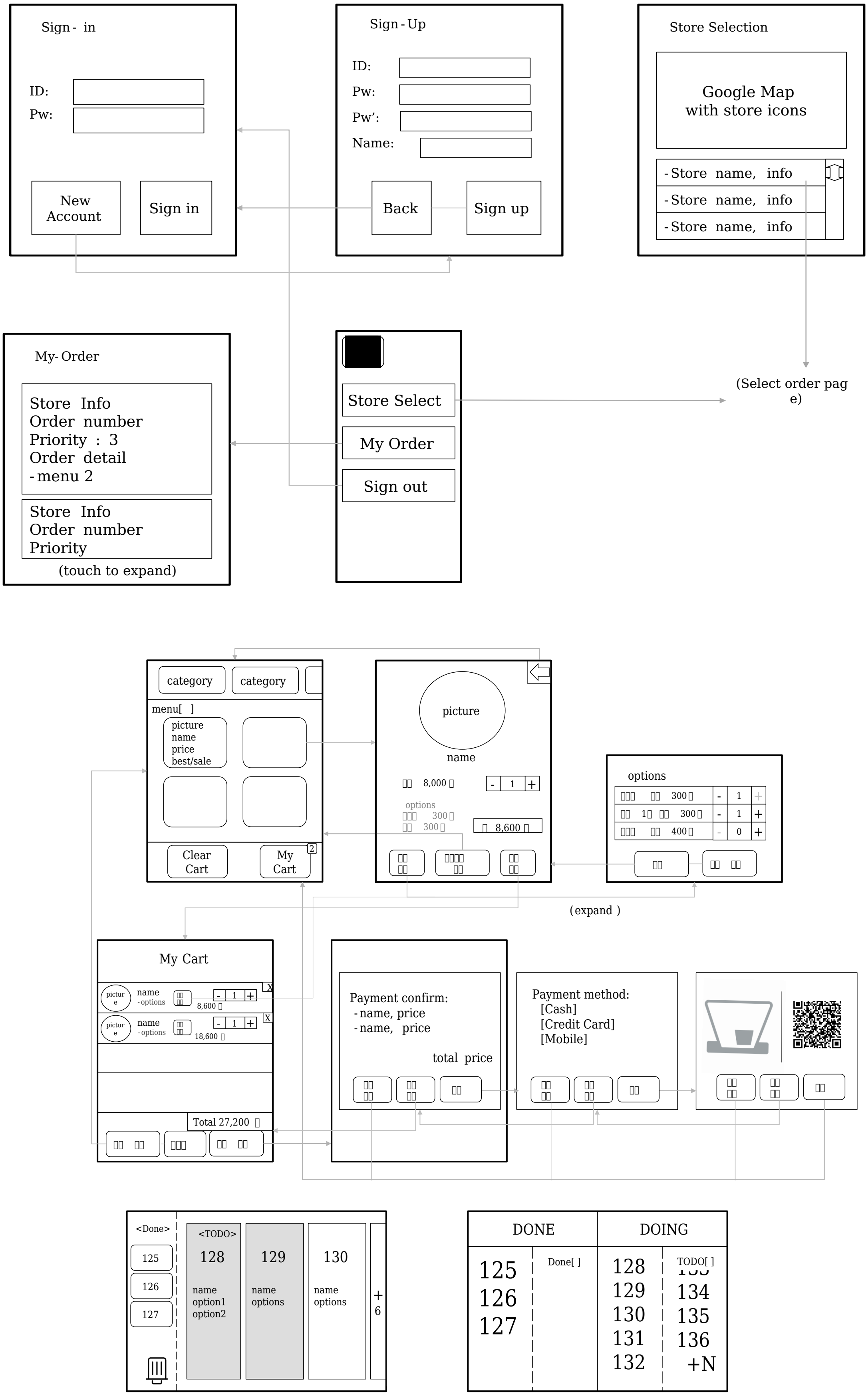
4.1.1. Model

This is django-specific model diagram. Actually, since only models we deal with in development are Django ORM ones, we described it in class-like diagram rather than relational UML. (A -o B) means A to B is one to many relationship and (A o-o B) means A to B is many to many relationship.


TxItemOption
id AutoField
purchasable_option ForeignKey (id)
tx_item ForeignKey (id)
base_price DecimalField
purchasable_option_name CharField
qty IntegerField
total_price DecimalField
Session
<AbstractBaseSession>
session_key CharField
expire_date DateTimeField
session_data TextField
abstract inheritance
AbstractBaseSession
expire_date DateTimeField
session_data TextField
tx_item (txitemoption)
TxItem
id AutoField
purchasable ForeignKey (id)
tx ForeignKey (id)
tx_log ForeignKey (id)
price DecimalField
purchasable_base_price DecimalField
purchasable_name CharField
qty IntegerField
total_price DecimalField
tx_log (txitem)
TxLog
<TxLike>
id AutoField
store ForeignKey (id)
tx ForeignKey (id)
user ForeignKey (id)
created_at DateTimeField
extra_props TextField
part_ref TextField
purchase_data TextField
purchase_type CharField
reversed BooleanField
state CharField
total_price DecimalField
updated_at DateTimeField
utxid CharField
TxCredit
id AutoField
tx ForeignKey (id)
amount DecimalField
created_at DateTimeField
customer_bank CharField
customer_name CharField
customer_ref CharField
reverse BooleanField
updated_at DateTimeField
value_date DateField
tx (txitem)
Ticket
<UserOwnable>
id AutoField
store ForeignKey (id)
tx ForeignKey (id)
user ForeignKey (id)
created_at DateTimeField
denorm_data TextField
number IntegerField
removed BooleanField
state CharField
updated_at DateTimeField
tx (txlog)
tx (txcredit)
tx (ticket)
Tx
<TxLike>
id AutoField
store ForeignKey (id)
user ForeignKey (id)
created_at DateTimeField
extra_props TextField
part_ref TextField
purchase_data TextField
purchase_type CharField
reversed BooleanField
total_price DecimalField
updated_at DateTimeField
utxid CharField
abstract inheritance
abstract inheritance
store (ticket)
TxLike
<UserOwnable>
store ForeignKey (id)
user ForeignKey (id)
created_at DateTimeField
extra_props TextField
part_ref TextField
purchase_data TextField
purchase_type CharField
reversed BooleanField
total_price DecimalField
updated_at DateTimeField
utxid CharField
purchasable_option (txitemoption)
purchasable (txitem)
store (txlike)
Store
<UserOwnable>
id AutoField
franchise ForeignKey (id)
user ForeignKey (id)
location PointField
name CharField
next_number IntegerField
timezone TextField
purchasable_categories (store)
PurchasableCategory
<UserOwnable,FranchiseOwnable>
id AutoField
franchise ForeignKey (id)
user ForeignKey (id)
name CharField
abstract inheritance
purchasables (purchasablecategory)
Purchasable
<UserOwnable,FranchiseOwnable>
id AutoField
franchise ForeignKey (id)
user ForeignKey (id)
base_price DecimalField
image ImageField
name CharField
abstract inheritance
abstract inheritance
franchise (store)
abstract inheritance
purchasable_options (purchasable)
abstract inheritance
PurchasableOption
id AutoField
base_price DecimalField
max_capacity IntegerField
name CharField
FranchiseOwnable
franchise ForeignKey (None)
abstract inheritance
abstract inheritance
franchise (franchiseownable)
Franchise
<UserOwnable>
id AutoField
user ForeignKey (id)
name CharField
abstract inheritance
LogEntry
id AutoField
content_type ForeignKey (id)
user ForeignKey (id)
action_flag PositiveSmallIntegerField
action_time DateTimeField
change_message TextField
object_id TextField
object_repr CharField
UserOwnable
user ForeignKey (id)
user (logentry)
user (userownable)
User
<AbstractUser>
id AutoField
date_joined DateTimeField
email EmailField
first_name CharField
is_active BooleanField
is_staff BooleanField
is_superuser BooleanField
last_login DateTimeField
last_name CharField
password CharField
username CharField
groups (user)
abstract inheritance
AbstractUser
<AbstractBaseUser,PermissionsMixin>
date_joined DateTimeField
email EmailField
first_name CharField
is_active BooleanField
is_staff BooleanField
is_superuser BooleanField
last_login DateTimeField
last_name CharField
password CharField
username CharField
content_type (logentry)
user_permissions (user)
Group
id AutoField
name CharField
permissions (group)
abstract inheritance
abstract inheritance
AbstractBaseUser
PermissionsMixin
Permission
id AutoField
content_type ForeignKey (id)
codename CharField
name CharField
content_type (permission)
ContentType
id AutoField
app_label CharField
model CharField

4.1.2. View

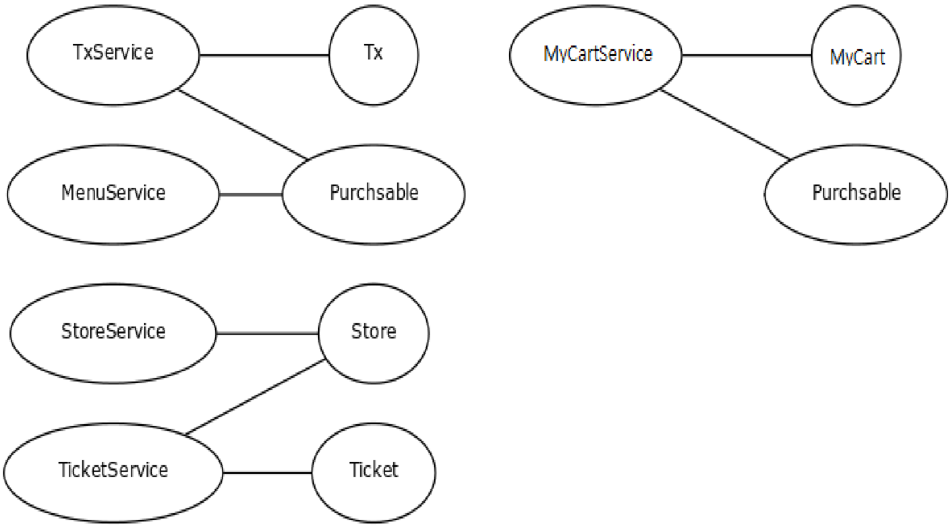
Brief view relationship is as follows:



For detailed explanations about each pages, see Design and Planning Document.

4.1.3. Service

Services are shown below:

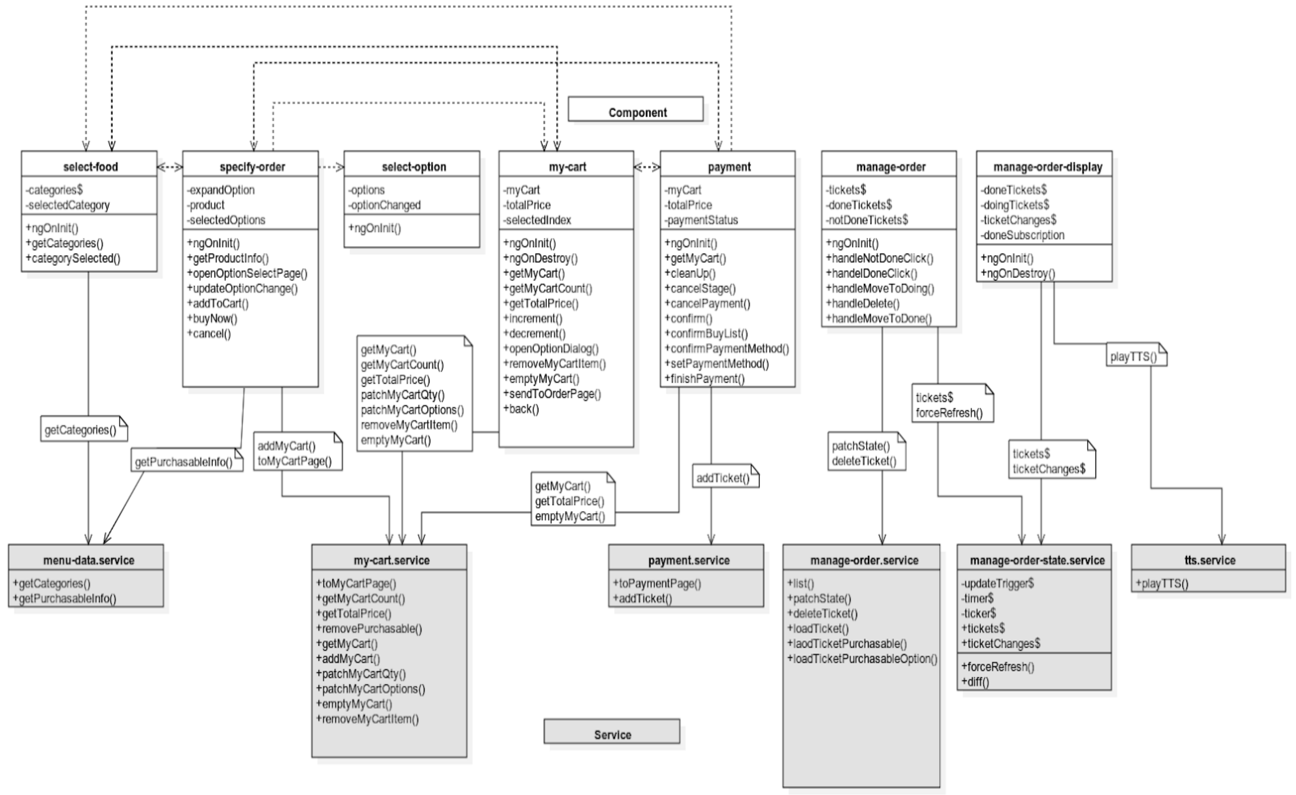


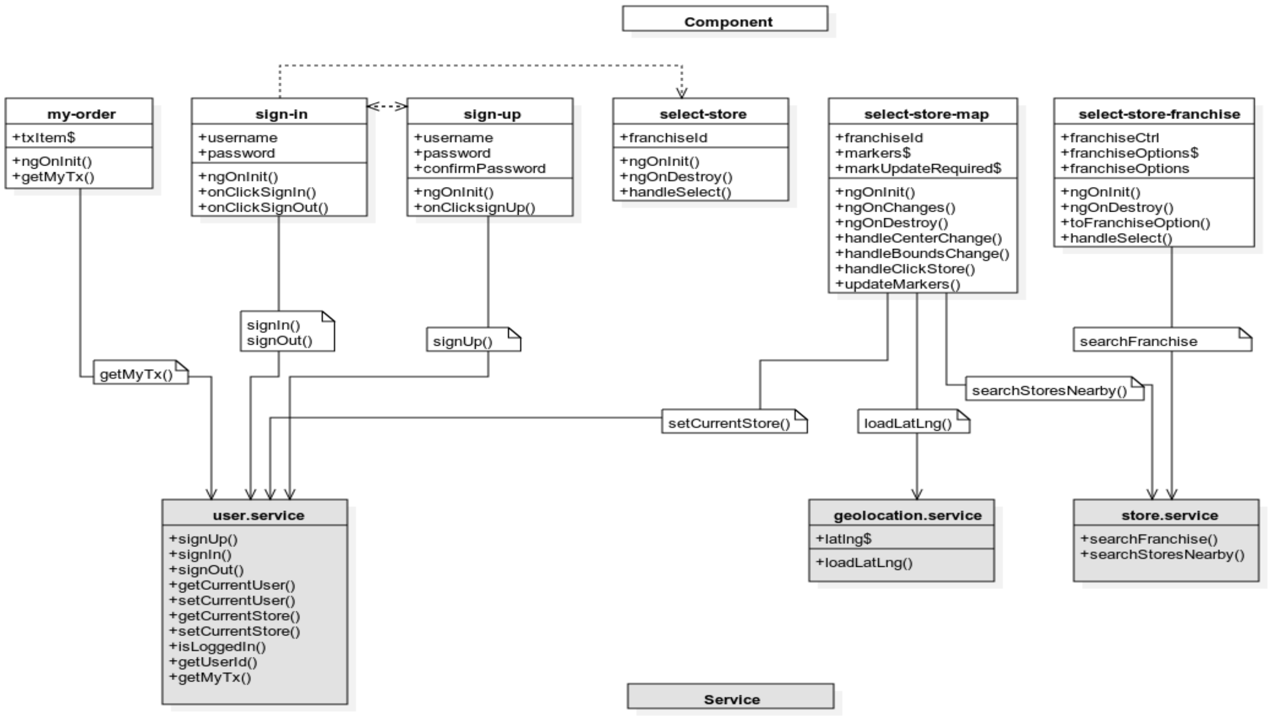
Services are glues between models and views. As shown above, services not only connect between models, but they also imply domain logic that TxService requires both Tx model and Purchasable model to determine how much user should pay and which purchasable we need to bookkeep in a transaction. Similar roles applied to MyCartService and MyCart model. More details will be discussed in *Design Details* section.

4.2. Frontend & Backend Design

4.2.1. Frontend

Below are our frontend design structure. (Relations among components and services)





4.2.2. Backend

We used REST API to communicate between backend and frontend. API endpoints for our application are:

Model	API	GET	POST	PUT	DELETE	PATCH
User	api/v1/sign_in	x	Provide an access key	x	x	x
	api/v1/sign_out	x	Discard an access key	x	x	x
Franchise	api/v1/franchise	Find franchise by name	x	x	x	x
Store	api/v1/store	Find nearby stores	x	x	x	x
Purchasable	api/v1/purchasable	Get menus of purchasables	x	x	x	x
	api/v1/purchasable/:id	Get specific purchsable	x	x	x	x
Tx	api/v1/tx	Get a transaction info	Build a new transaction	x	x	x
	api/v1/tx/:id/finish	x	Finish a transaction	x	x	x
Ticket	api/v1/ticket	Get all tickets of my store	x	x	x	x
	api/v1/ticket/:id	Get specific ticket state	x	x	Delete a ticket (when done)	Change ticket state
Sound URL(TTS)	api/v1/order_tts	Get a sound url for noticing a customer his/her order is ready	x	x	x	x

5. Implementation Details

5.1. Brief Development Process

KiWi is implemented with python3.7 for backend, angular 6 for frontend. Django is used as web framework. For front-end testing, jasmine is utilized as it is default option for it. Unlike we have exercised in practice session and final exam, we alternately adopt pytest framework rather than plain-old unittest module. Finally, the whole web application is deployed to Heroku, which is easy-to-use PaaS. Every app version is contained into a docker container, uploaded to heroku container service, and then released to heroku service.

5.2. Databases, Libraries, Frameworks, and External Services

Databases are the fundamental source of data of all SaaS. Kiwi is not an exception for this. Additionally, we require database specialized for geolocational query as we have store location search as core function. Fortunately, thanks to django ORM and djangoGIS abstraction, we don't have to worry about underlying GIS database structure because ORM normalizes all queries and abstract out every detail needed to execute query. So databases differ only in database performance characteristics, the way to store and where to store data. For agile development, we thought sqlite3 with GIS extension was the best choice for this workload. So initially development were done with it. Deploying to real server in wild, we were frustrated to use sqliteGIS because only serve database for the same file system, not through network. This was a critical issue because all SaaS server tends to be volatile: all servers can be born and shut down many times a day. So we finally replaced sqlite by PostGIS, posgresql variant for geolocation query. Fortunately, we didn't have any issue during transition thanks to strong abstraction that django ORM provides.

Python dependencies of project are managed with `pipenv`. Comparing to traditional approach utilizing `pip`, `virtualenv`, and maintaining `requirements.txt`, it provides all-in-one package manager that combines those three roles. Plus, it can lock the dependencies into `Pipfile.lock` so any version of external library can be verified and fixed.

Although `npm` is widely adopted package manager for nodejs, it is notorious for slowness. As we are lazy coders, we chose to use `Yarn`, drop-in replacement package manager of npm, which shines the light when performance matters.

Major libraries used are followings:

- gtts: provides Google Text-To-Speech service
- uwsgi: application server
- django-map-widget: provides Google Map integration in django admin
- psycopg2: PostGIS connector in python
- @angular/material: Material CSS framework for angular

5.3. Testing and CI

Our codebase is divided into two ends: frontend, backend. For each unit, it is split into sublayers to be tested. In order to carry out unit-testing time efficiently, mocks and stubs are required to be installed in place of their dependencies.

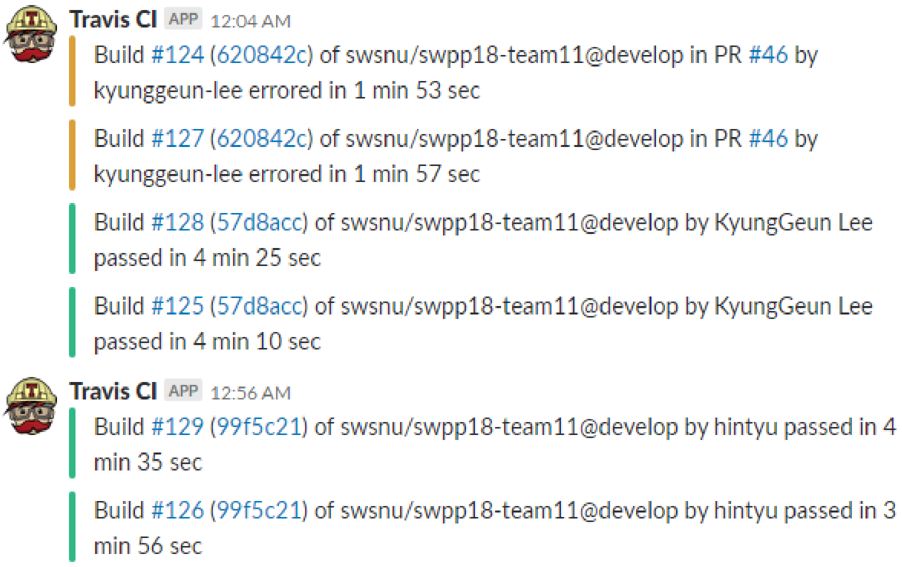
In our design, backend server consists of three layers: models, services, and views. Model classes, which are all located in models/ directory, are tested with `sqlite3 memory db`. Services are glues between models and views. We will be utilizing `MagicMock` class and `patch` function from standard `unittest.mock` library. Likewise, Testing on views will be covered with similar manner. We chose `pytest` instead of python unittest module, because of clear syntax and easy fixture declaration.

All APIs are covered. Also we tried to test doubles as possible. Mocking (or testing doubles) made testing fast and allowed for more modular code structure, although not all database use cannot be replaced with mock as django authentication module is tightly coupled with session and user model, which inevitably makes access to database. As follow is an example of use of mocking to reduce time for unit test where `TxService` is stubbed so as to test only the functionalities between API and service and not dig into service and database access, which is the common foremost reason for slow tests.

```
def test_create_finish_tx(user_logged_in, Store, client):
    order_tx = MagicMock(name="OrderTx")

    with patch('kiorder.api.v1.test_tx.TxService') as TxService:
        TxService().load.return_value = order_tx
        response = client.post('/kiorder/api/v1/test_tx/UTXID/finish')
        assert response.status_code == 200

        TxService().load.assert_called_with("UTXID")
        TxService().finish_order.assert_called_with(order_tx)
```

Behavior driven design could not be practiced because of lack of experience on it. Like TDD, it demands experience to some degree on unit testing philosophy and testing framework being used. Besides, it was not so joyful to practice BDD or TDD as a newcomer of web development. We thought it is the most important thing to enjoy development process to be motivated.

It is critical to ensure all unit test suite pass. Otherwise, it renders unit testing pointless because testing can't be indicator that codes work well anymore. But, as time passed, we found it hard to keep 100%-all green to commit rule in the middle of project. Because project participants were easy to forget unit testing before commit or oblivious to keeping the rule of all-tests-being-green for everyone of our team. To keep testing guideline, we employed Travis CI to enforce rules. Every commit and pull request passes through CI so they are tested with several guideline: coding style, front-end test, backend-tests, and coverage report. The results of CI are reported onto both GitHub PR page and Slack chat room accordingly.



Travis CI APP 12:04 AM

Build #124 (620842c) of swsnu/swpp18-team11@develop in PR #46 by kyunggeun-lee errored in 1 min 53 sec

Build #127 (620842c) of swsnu/swpp18-team11@develop in PR #46 by kyunggeun-lee errored in 1 min 57 sec

Build #128 (57d8acc) of swsnu/swpp18-team11@develop by KyungGeun Lee passed in 4 min 25 sec

Build #125 (57d8acc) of swsnu/swpp18-team11@develop by KyungGeun Lee passed in 4 min 10 sec

Travis CI APP 12:56 AM

Build #129 (99f5c21) of swsnu/swpp18-team11@develop by hintyu passed in 4 min 35 sec

Build #126 (99f5c21) of swsnu/swpp18-team11@develop by hintyu passed in 3 min 56 sec

5.4. Deployment

Heroku was the best choice of PaaS for us because it provides Postgresql database plugin and memcached plugin at zero cost. Plus, it supports docker container for deployment option.

For each service, frontend and backend, we wrote Dockerfiles to build a container. Then, we deployed them to https://swpp-blender-frontend-staging.herokuapp.com, https://swpp-blender-backend-staging.herokuapp.com, respectively.

Dockerfile of frontend sets up from nginx image, build angular app to bundle js file and copy it to public directory of nginx from which files are served.

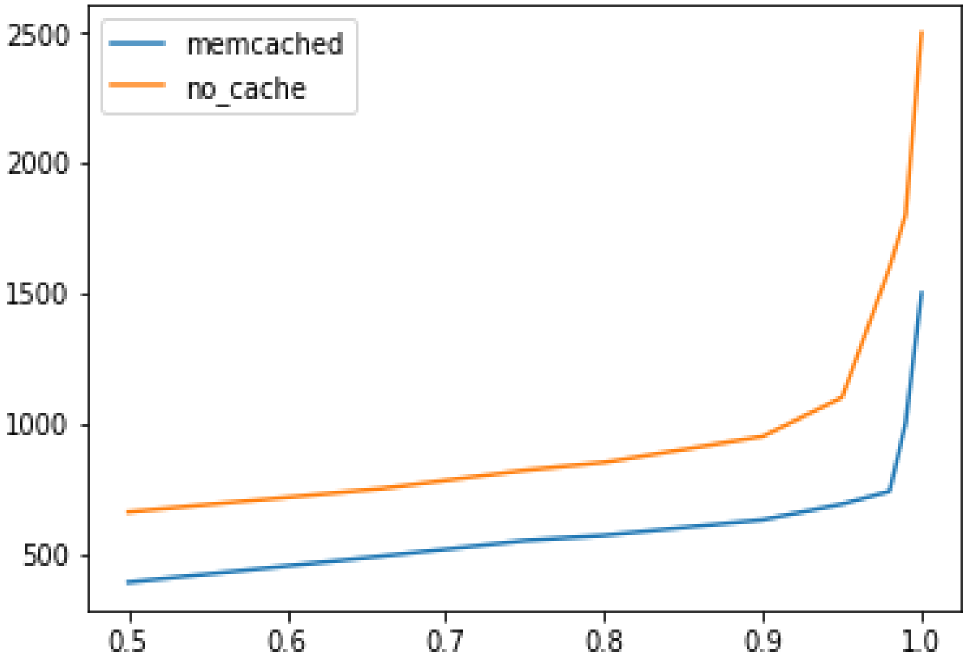
```
FROM nginx

WORKDIR /usr/src/app
RUN apt-get update && apt-get install -y curl gnupg && curl -sL https://deb.nodesource.com/setup_8.x | bash -
RUN apt-get update && apt-get install -y nodejs gettext-base
RUN npm install --global yarn && npm upgrade --global yarn
COPY src/ src/
COPY angular.json .
COPY package.json .
COPY tsconfig.json .
RUN yarn && yarn build --prod

RUN mkdir -p /usr/share/nginx/html
RUN cp -r dist/* /usr/share/nginx/html/

COPY deployment/nginx.conf /etc/nginx/nginx.conf.template

CMD envsubst '$PORT $API_DOMAIN' < /etc/nginx/nginx.conf.template | tee /etc/nginx/nginx.conf && nginx
```

Dockerfile of backend gets all requirements in `Pipfile.lock` installed, collects all static files like css and js, and then run uwsgi server.

```
FROM python:3.7.1-stretch

EXPOSE 3031
VOLUME /usr/src/app/kiwi/media
VOLUME /usr/src/app/kiwi/database
WORKDIR /usr/src/app

RUN pip install pipenv
RUN apt-get update && apt-get install -y curl libtiff5-dev libjpeg62-turbo-dev zlib1g-dev \
        libproj-dev libgeos-dev libspatialite-dev libsqlite3-mod-spatialite gdal-bin \
        libfreetype6-dev liblcms2-dev libwebp-dev tcl8.6-dev tk8.6-dev python-tk python3-dev

COPY Pipfile Pipfile
COPY Pipfile.lock Pipfile.lock
RUN pipenv install --system --deploy --ignore-pipfile

COPY . .
RUN cd kiwi && ./manage.py collectstatic --no-input
```

```
CMD cd kiwi && uwsgi \
    --processes 4 \
    --threads 3 \
    --wsgi-file kiwi/wsgi.py \
    --http :$PORT
```

5.5. Performance



https://www.notion.so/ba4e9f77e5694086a7b7f58b27f3b314

For the sake of performance, we employed API cache for menu and purchsables and saw significant performance gain. To see the effect of cache, latency of API backend server has been measured. Using `locust.io`, the web load test framework, we simulated a scenario where 10 users concurrently connect to server and send requests at a rate of 2 requests per second.

As a result, we saw a significant increase of performance comparing API server with memcached to one without it. For all percentile, we see 20% latency reduction . Even though we added a cache to one API endpoint, which was only feasible target to cache, we found overall performance has increased. We suspect that this is because worker threads can finish an API request earlier just to process next request in queue.

6. Lessons Learned

- **Techniques and Concepts of Software Development**
  Throughout this project, we could learn techniques and concepts of software development. This includes: 1. How to use frameworks such as Angular and Django 2. How to test 3. Iterative process of software development(AGILE) 4. Bird's-eye view on service architecture 5. Principles for software development.

- **Impact of Database on Overall Performance**
  As we mentioned above, we employed API cache in backend and the result was very successful, even though we cached only a fraction of API requests. Such an observation reveals that the impact of caching database on the performance of the whole system is paramount.

- **Difficulty of Commercialization**
  Although we put special focus on the commercial value of our service, it turned out that developing a key idea into a real world commercial application is almost always more complex than our expectation. One such example was that there could be various possible scenarios of usage such as tagging student id for discount.
  Besides, deciding details was one thing, and working on them was another. Most of our core features were finished before mid-term presentation, but we had to spent the rest of this semester mainly in implementing auxiliary (or "trivial", depending on view point) features. Indeed, the Pareto principle (or 80-20 rule) proved to be true again.

7. Conclusion

Our project, *KiWi Order*, is a kiosk ordering system with cost-efficiency and flexibility as main features. Our frontend design consists of component and service; each component provides a view of and control over a page, and services provide functionalities that multiple components share. Similarly, the backend design consists of model, service, and view; services provide functionalities for views. For implementation, we used frameworks like Angular 6 and Django, external libraries such as gtts, django-map-widget, etc., jasmine and pytest for testing, heroku for deployment, and other tools to keep our code clean and consistent. Along the course and project, we could obtain techniques and concepts for software development. We could also have a small experiment on the impact of database, achieving considerable improvement using API cache. Compared to the current solutions, *KiWi Order* are generally more cost-efficient and flexible, although there is some room for improvement in some detailed features. For future improvements, for example, introducing QR codes, adding manager's page, and order queuing algorithm for peak time could be considered.

8. Appendix: Sprint 6 Progress Report

8.1. Progress

- **Sign In, Sign Up features**
  We added sign-in and sign-up feature to allow customers to sign in their own account and order stuffs. In frontend, we used authguard (which checks authentications every time the routing happens) to prevent unauthorized users from visiting other pages using. In backend, we added some property login_required to some methods of views to block unauthorized requests.

- **Badge**
  Badge is a feature to attach badges(like 'best', 'sale', 'hot') next to Product names. Badge images are loaded from backend, and becomes one of purchasable's properties. Backend purchasable model contains information about the badges that purchasable would have.

- **Idle-timeout module**
  Idle-timeout uses angular-user-idle module to detect user idle status. If user doesn't make any movements for 60 seconds, user will be sent to the front page of store(select-food). This is a feature for kiosks to be installed in store.

- **Hamburger menu**
  Handy navigating tool to navigate around our web service. Can detect whether user is signed in or not, and has chosen store or not to differ the contents of hamburger menu.

- **Caching**
  We added memcache to purchasable model and measured the performance improvement using locust. Detail reports on 'Implementation Details - Performance'

- **Responsive web view and CSS**
  Added sample responsive web view at the front page of store(select-food). It shows menus as a list when display's width is below 768 pixels, and it shows menus as cards otherwise

8.2. Difficulties

- **Deployment problems - CORB and cookies**
  We chose to deploy our service on two separate heroku servers, as a frontend and backend server each. The reason for this was because if we deploy front- and backend on the same server, and if one of the two has to be upgraded and re-released, its counterpart also has to re-released, which degrades quality of service.

However, this choice caused some side effects including CORB and cookie issues. To be more specific, since the front- and backend are now in separate domains, the CORS policy has become a problem. Moreover, we had to do some extra work to solve the problem where cookies are not sent to backend along with request.

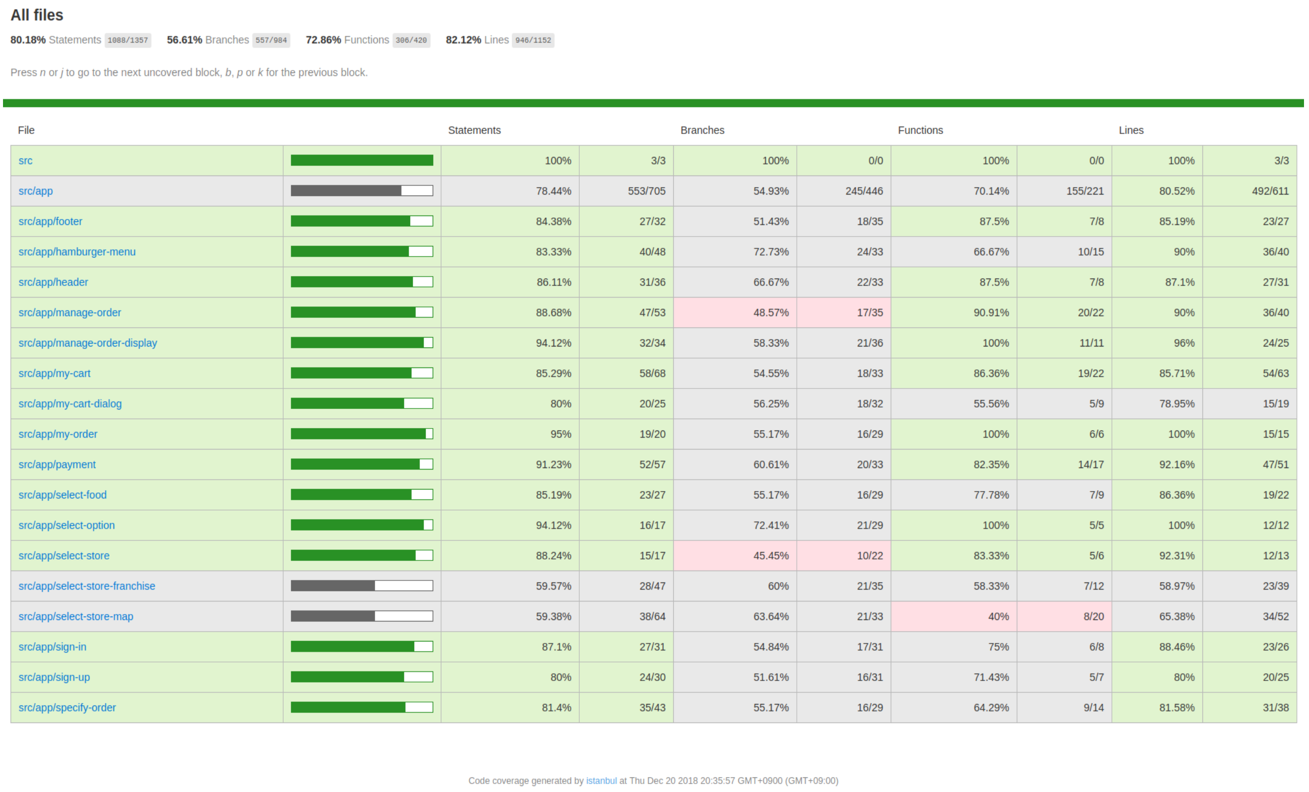
8.3. Test Coverage Report

The overall test coverage was approx. 80% for the frontend, and 90% for the backend. Note, however, that some components(frontend) or APIs(backend) are marking low coverage. The reason for this is mainly because those were so tightly coupled with external libraries, such as gtts, that it is either meaningless or difficult(or both) to test them exhaustively.

8.3.1. Test Objects

- **Backend**: same as previous sprint
- **frontend**: newly merged Sign-in, Sign-up, Hamburger Menu Components and Services

8.3.2. Frontend



All files

80.18% Statements 1088/1357 **56.61%** Branches 557/984 **72.86%** Functions 306/420 **82.12%** Lines 946/1152

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements		Branches		Functions		Lines	
src		100%	3/3	100%	0/0	100%	0/0	100%	3/3
src/app		78.44%	553/705	54.93%	245/446	70.14%	155/221	80.52%	492/611
src/app/footer		84.38%	27/32	51.43%	18/35	87.5%	7/8	85.19%	23/27
src/app/hamburger-menu		83.33%	40/48	72.73%	24/33	66.67%	10/15	90%	36/40
src/app/header		86.11%	31/36	66.67%	22/33	87.5%	7/8	87.1%	27/31
src/app/manage-order		88.68%	47/53	48.57%	17/35	90.91%	20/22	90%	36/40
src/app/manage-order-display		94.12%	32/34	58.33%	21/36	100%	11/11	96%	24/25
src/app/my-cart		85.29%	58/68	54.55%	18/33	86.36%	19/22	85.71%	54/63
src/app/my-cart-dialog		80%	20/25	56.25%	18/32	55.56%	5/9	78.95%	15/19
src/app/my-order		95%	19/20	55.17%	16/29	100%	6/6	100%	15/15
src/app/payment		91.23%	52/57	60.61%	20/33	82.35%	14/17	92.16%	47/51
src/app/select-food		85.19%	23/27	55.17%	16/29	77.78%	7/9	86.36%	19/22
src/app/select-option		94.12%	16/17	72.41%	21/29	100%	5/5	100%	12/12
src/app/select-store		88.24%	15/17	45.45%	10/22	83.33%	5/6	92.31%	12/13
src/app/select-store-franchise		59.57%	28/47	60%	21/35	58.33%	7/12	58.97%	23/39
src/app/select-store-map		59.38%	38/64	63.64%	21/33	40%	8/20	65.38%	34/52
src/app/sign-in		87.1%	27/31	54.84%	17/31	75%	6/8	88.46%	23/26
src/app/sign-up		80%	24/30	51.61%	16/31	71.43%	5/7	80%	20/25
src/app/specify-order		81.4%	35/43	55.17%	16/29	64.29%	9/14	81.58%	31/38

Code coverage generated by istanbul at Thu Dec 20 2018 20:35:57 GMT+0900 (GMT+09:00)

8.3.3. Backend

```
Name                                      Stmts   Miss  Cover
-------------------------------------------------------------
kiorder/api/__init__.py                       1      0   100%
kiorder/api/errors.py                         6      0   100%
kiorder/api/v1/__init__.py                    1      0   100%
kiorder/api/v1/base.py                       63      4    94%
kiorder/api/v1/franchise.py                  14      0   100%
```

```
kiorder/api/v1/my_cart.py                   63      9    86%
kiorder/api/v1/purchasable.py               31      0   100%
kiorder/api/v1/store.py                     30      6    80%
kiorder/api/v1/test_tx.py                   41      5    88%
kiorder/api/v1/ticket.py                    34      0   100%
kiorder/api/v1/user.py                      66     45    32%
kiorder/models/__init__.py                  16      0   100%
kiorder/models/badge.py                      6      1    83%
kiorder/models/base.py                       0      0   100%
kiorder/models/franchise.py                  6      1    83%
kiorder/models/mixins.py                    10      0   100%
kiorder/models/my_cart.py                    5      0   100%
kiorder/models/my_cart_item.py               7      0   100%
kiorder/models/my_cart_item_option.py        7      0   100%
kiorder/models/purchasable.py               13      1    92%
kiorder/models/purchasable_category.py       9      1    89%
kiorder/models/purchasable_option.py         7      1    86%
kiorder/models/store.py                     14      1    93%
kiorder/models/ticket.py                    16      0   100%
kiorder/models/tx.py                         9      1    89%
kiorder/models/tx_credit.py                 12      0   100%
kiorder/models/tx_item.py                   13      0   100%
kiorder/models/tx_item_option.py            10      0   100%
kiorder/models/tx_like.py                   24      0   100%
kiorder/models/tx_log.py                     8      0   100%
kiorder/models/user.py                      11      0   100%
kiorder/services/__init__.py                 0      0   100%
kiorder/services/franchise.py                5      0   100%
kiorder/services/menu.py                    37      0   100%
kiorder/services/my_cart.py                 54      4    93%
kiorder/services/store.py                   23      0   100%
kiorder/services/ticket.py                  29      0   100%
kiorder/services/tx.py                     165      8    95%
kiorder/services/utils.py                    3      0   100%
------------------------------------------------------------
TOTAL                                      869     88    90%
```