Testing Plan of ERP-team12

Tyler Znoj 4005987
Killian Kelly 40014508
Abdul Sirawan 40074202
Adrien Kamran 40095393
Qandeel Arshad 40041524
Cedrik Dubois 40064475
Anthony Chraim 40091014
Matthew Giancola 40019131

A report submitted in partial fulfilment of the requirements of SOEN 390 Concordia University

Date 3/1/2021

Testing Procedure

1. General Approach

The main approach which will be adopted for this project is Unit testing of modules. The application is a web-based application and it is developed using Django, which is a high-level Python web framework. Django has a test-execution framework and assorted utilities which allows us to simulate requests, insert test data, inspect the output of the application and generally verify if the code is doing what it should be doing. The preferred way to write tests in Django is using the **unittest** module built-in to the Python standard library.

For the purpose of this project, we will rely on the use of unittest throughout the entire development phase. Also, according to the requirements, the system shall have at least 50% test coverage in Controllers classes. The tool which will assess the test coverage of our application is called **django-nose**. It is a django coverage testing tool that displays the stats of our testing coverage.

1.1 Bugs Handling

I. Usage of SDK

Our first approach in handling bugs is using SDK (Software Development Kit) in the development process. Pycharm and Visual Studio Code are used for this project. They facilitate the creation of applications by having a compiler, and debugger. Their debugging system is extremely useful in identifying errors and bugs existing in the system. They also display a message indicating the source and the cause of the error.

II. Communicating the bugs

Bugs consume an incredible amount of time from the development team in efforts of resolving them. However, this wasted time can be put into a better usage if the bug is killed quicker. One of the approaches our team is adopting is communicating bugs with the team members first. There is a high chance that another team member has encountered the same bug and knows the solution, because we are eventually working on the same project and using similar development tools. Also, working collaboratively in resolving a bug is much faster than researching and reading documentation individually. Consequently, bugs are communicated with the rest of the team members upon their emergence. And whoever is experiencing the same bug, they partner up together and work towards resolving the bug.

III. Classifying the bugs

When a bug emerges, the team will conduct a meeting to discuss its severity and classify it whether one of the following:

a) High-severity

The quality assurance team shall evaluate the bug to high-severity if it seems that it hinders the functionality of the program. The program must work in compliance with the functional requirements. Therefore, when a bug occurs during the development of a feature from the functional requirements, it should be mitigated promptly and perform unit testing to verify the functionality of the program.

b) Medium-severity

The quality assurance team shall evaluate the bug to medium-severity if it impacts the performance and security of the program. After implementing a functional requirement feature, the QA team will analyze the security and performance of the program and ensure that it is in accordance with the requirements. Thus, any bug relevant to those two non-functional requirements.

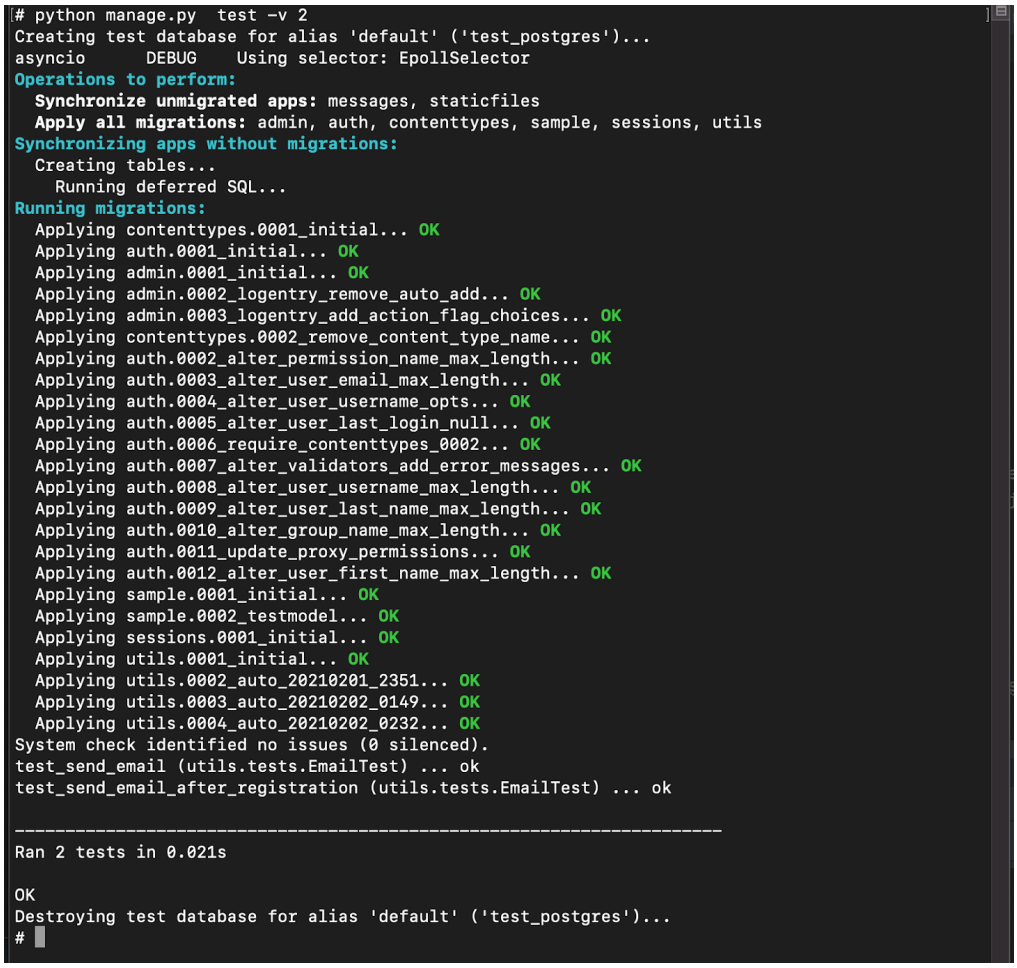
c) Low-severity

The quality assurance team shall evaluate the bug to low-severity if it is related to the UI of the application. Any bugs relevant to the design of the application will be fixed before submitting each sprint. Therefore, it is left until the end to be resolved after fixing bugs related to the functional and non-functional requirement.

2. Testing Approach

2.1 Unit Testing

The application shall be composed of many unit tests that test most of the functions in many modules, especially the controllers modules which should have at least 50% test coverage. The plan is that every developer has to write unit tests for the functionalities that he/she implemented, before pushing the changes to the repository. Later on, those unit tests will be evaluated by 2 selected members representing our quality assurance team. The test platform used will be the Django included test framework using the unittest module built-in the Python standard library. Below is an example of a test run using the tool:



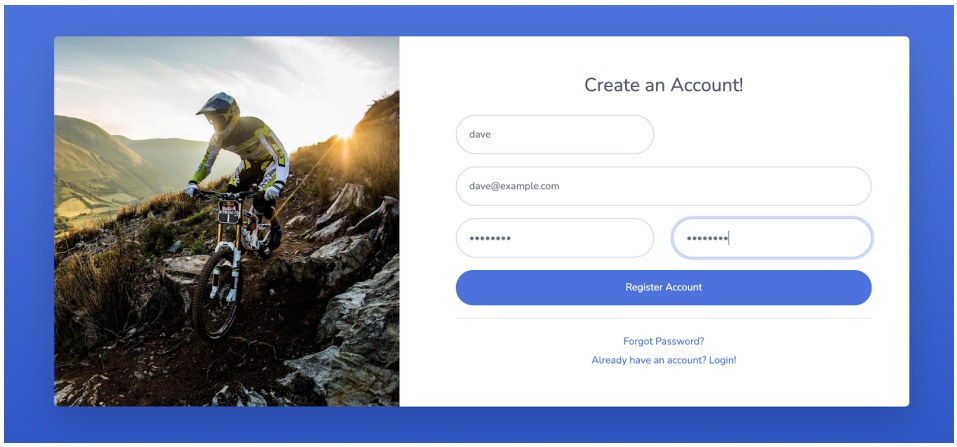
```
# python manage.py  test -v 2
Creating test database for alias 'default' ('test_postgres')...
asyncio      DEBUG    Using selector: EpollSelector
Operations to perform:
  Synchronize unmigrated apps: messages, staticfiles
  Apply all migrations: admin, auth, contenttypes, sample, sessions, utils
Synchronizing apps without migrations:
  Creating tables...
    Running deferred SQL...
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sample.0001_initial... OK
  Applying sample.0002_testmodel... OK
  Applying sessions.0001_initial... OK
  Applying utils.0001_initial... OK
  Applying utils.0002_auto_20210201_2351... OK
  Applying utils.0003_auto_20210202_0149... OK
  Applying utils.0004_auto_20210202_0232... OK
System check identified no issues (0 silenced).
test_send_email (utils.tests.EmailTest) ... ok
test_send_email_after_registration (utils.tests.EmailTest) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.021s

OK
Destroying test database for alias 'default' ('test_postgres')...
#
```

2.2 Continuous Integration Test

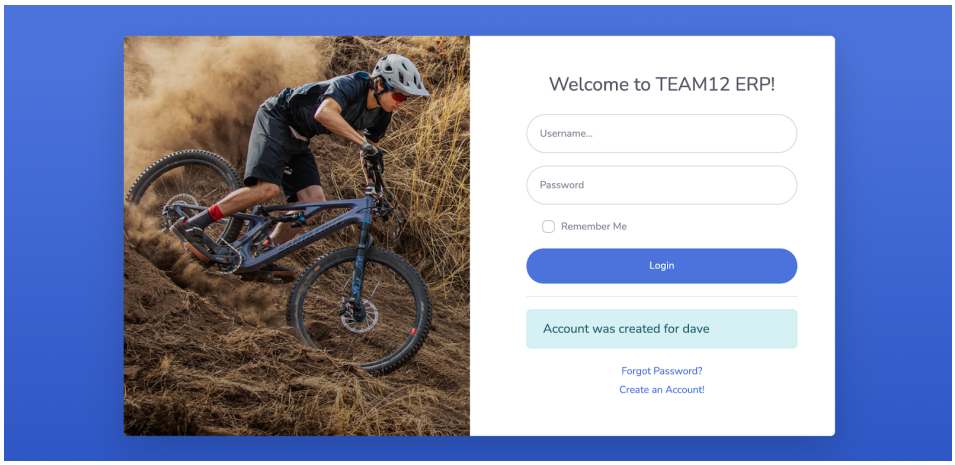
To ensure that new features introduced do not affect the current state of the application, continuous integration tests will be performed on each pull request before it is reviewed. This will be done using CircleCi, a free integration testing platform. This will be connected to the github repository and will set up the correct environment necessary and perform all unit tests of the application.

2.3 Email Test

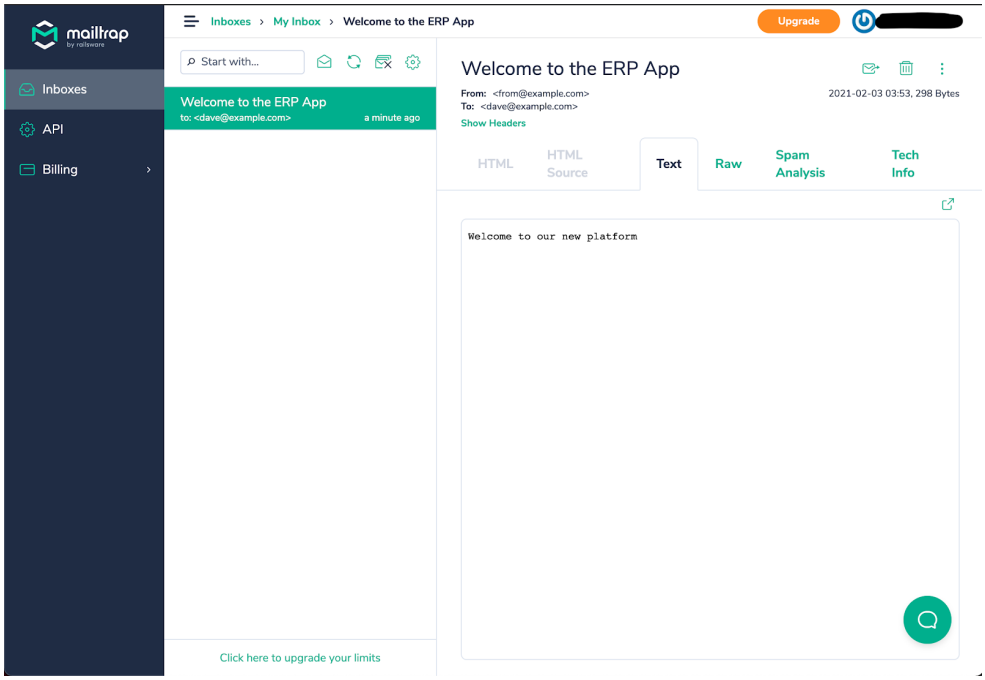
For functionality concerning emails, Mailtrap will be used. It provides a service for the safe testing of emails sent from the development and staging environments. Mailtrap catches the email in a virtual inbox to analyse the proper functionalities concerning emails. It provides a SMTP server and configuration necessary for this. It will also be used to render any HTML email to check if the proper rendering and check the email template for responsiveness. Mailtrap can also be used in the automatic continuous integration tests with their API. An example is shown below:



A user named dave registers for an account.



dave successfully creates a new account.



An email is sent to his email to welcome him.

3. Communication Approaches

3.1 Weekly meeting

The QA assurance team which is composed of 2 selected members will perform a weekly meeting between each other to discuss the code and the tests submitted by the development team. The QA assurance team will start with the controllers classes as they pose the greatest risk because they control the functionality of the program. The QA team shall start the process by performing coverage tests for all controllers modules and evaluate the states. If a controller module has lower than 50% coverage, it is investigated further to decide on different tests to test the functionality of the module. The missing unit test is added and ensured that major functionalities of the controller modules are tested exhaustively.

Moreover, the QA team will assess the tests written for the rest of modules and evaluate if they are efficient and necessary. In the next scheduled meeting of the development team, the QA team will be assigned a slot of time in the meeting to present their findings to the other team members.

Sprint2 Testing Procedure

1. General Approach

The procedure was to use Python standard library module **unittest** to test the forms that we use in validating and communicating data. Along with the urls file where we direct the visited urls to the right page. The Controller module for our app was the views.py file where all the bank-end logic is handled.

1. Testing forms

Inside the testing form, we had a form that processes the user information during login and registration. This form is imported from a library that Django provides. It also has validating parameters while processing the values. So, we tested major different possibilities that a user may enter values in that form. All of our tests check out to be a success. The form in fact validates the proper entry of the email, password, repeat password fields. Here's a code snippet of one of the validation test that was written on an invalid email:

```
def test_form_invalid_email(self):
    form1 = CreateUserForm()
    data1 = {
        'username': 'testingUser',
        'email': 'notValid',
        'password1': 'test123456!',
        'password2': 'test123456!',
    }
    form1 = CreateUserForm(data1)
    self.assertFalse(form1.is_valid())
```

And here's the result for that test:

```
test_form_invalid_email (sample.tests.test_forms.TestCreateUserForm) ... ok
```

2. Testing urls

Inside the testing urls, we tested that our app handles the urls that are entered and makes calls to proper functions in the controller model that handles the processing the proper information and displaying results. Here's a code snippet testing that when the home url is entered our app resolves it to the home function:

```
def test_home_url_is_resolved(self):
    url = reverse ('home')
    self.assertEquals(resolve(url).func, home)
```

And here's the result for that test:

```
test_home_url_is_resolved (sample.tests.test_urls.TestUrls) ... ok
```

3. Testing views (Controller model)

Testing views was done in a detailed way to capture that the response to the POST and GET request is correct. Thus, each function we have in the views model is responsible for handling a POST/GET request. If a function receives both requests, the response of both requests is being checked. For example, we tested if a user enters valid information he/she should be redirected to the login page where they should login with the registered credentials. We have checkpoints to make sure that the right HTTP response code is returned and the right template is rendered. Here's the code snippet demonstration the above:

```
self.validUser = {
    'username': 'testingUser',
    'email': 'testing@gmail.com',
    'password1': 'test123456!',
    'password2': 'test123456!',
}
```

```
def test_register_POST_validUser(self):
    response = self.client.post(self.register_url, self.validUser, format='text/html')
    self.assertEquals(response.status_code, 302)
    self.assertRedirects(response, '/login')
```

And here's the result for that test:

```
test_register_POST_validUser (sample.tests.test_views.TestViews) ... sample.views DEBUG    creating account with name: testingUser
ok
```

If the user above enters invalid credentials he should be redirected to the same registration page until he enters valid credentials.

2. Test Coverage

Initially, it was decided that we will use **django-nose** to test coverage of our tests. However, during the installation process it was noticed that there exists a faster framework that tests coverage for our Django project and can generate an html report. The objective was to have more than 50% on the controller classes. However, we also aimed to test other relevant classes. Our app scored 90% test coverage on all classes. And 59% on the views class. However, we can also consider urls, forms, and decorators to be part of the controller classes. Here's the testing coverage report for our classes. The urls, forms, and decorators scored 100%, 100%, 88%, respectively. We were feeling really confident about our testing coverage because it indicates our written tests have high quality.

However, it is worth mentioning that those results are produced prior to refactoring of our code. The refactoring phase was done at the end of the sprint. It was mainly to go over missing white spaces, comments, and other code quality attributes. Thus, in the process it was realized that some of our models needed to be re-transformed and some methods had to be moved around. This process caused a lot of bugs and prevented testing of the proper functionality. However, It was planned that before sprint3, our code would run and achieve the objectives that were set.

Below is a screenshot of our test coverage prior to refactoring:

Coverage report: 90%

Module ↑	statements	missing	excluded	coverage
ERP/__init__.py	0	0	0	100%
ERP/settings.py	31	0	0	100%
ERP/urls.py	4	0	0	100%
inventory/__init__.py	0	0	0	100%
inventory/admin.py	1	0	0	100%
inventory/migrations/0001_initial.py	6	0	0	100%
inventory/migrations/__init__.py	0	0	0	100%
inventory/models.py	34	0	0	100%
inventory/tests.py	1	0	0	100%
manage.py	12	2	0	83%
sample/__init__.py	0	0	0	100%
sample/admin.py	1	0	0	100%
sample/decorators.py	8	1	0	88%
sample/forms.py	9	0	0	100%
sample/migrations/0001_initial.py	5	0	0	100%
sample/migrations/0002_testmodel.py	4	0	0	100%
sample/migrations/__init__.py	0	0	0	100%
sample/models.py	7	0	0	100%
sample/tests/__init__.py	0	0	0	100%
sample/tests/test_forms.py	27	0	0	100%
sample/tests/test_urls.py	28	0	0	100%
sample/tests/test_views.py	81	0	0	100%
sample/urls.py	3	0	0	100%
sample/views.py	103	42	0	59%
utils/__init__.py	1	0	0	100%
utils/apps.py	5	0	0	100%
utils/migrations/0001_initial.py	7	0	0	100%

Sprint3 Testing Procedure

1. General Approach

The procedure was to follow the same logic that was adopted in sprint2 for testing. Each application should have 3 tests files; test_urls.py, test_forms.py, and test_views.py. For this sprint, a lot of refactoring was done where each module will have its own app and its own tests.

2. Coverage Report

For this sprint, our total coverage percentage dropped from 90% to 73%. This is due to the addition of many major features and modules for this sprint. Those features included, but not limited to, tracking defects, tracking costs, and tracking orders. Those features require many lines of code. Moreover, our tests focused mainly on handling the urls entered and directing them to the right function. Also, each form used was tested and made sure that its fields are filled in with the proper information. Finally, the views classes were tested to ensure that when we submit a get or a post request we return the right information.

However, since a lot of logic lays behind the views classes. The views classes had the lowest coverage percentage and it was noted that those classes need to be tested further to increase our confidence in the robustness of the code.

Below is the coverage report for this sprint:

Coverage report: 73%

Module ↑	statements	missing	excluded	coverage
ERP/__init__.py	0	0	0	100%
ERP/settings.py	31	0	0	100%
ERP/urls.py	4	0	0	100%
accounting/__init__.py	0	0	0	100%
accounting/admin.py	3	0	0	100%
accounting/migrations/0001_initial.py	5	0	0	100%
accounting/migrations/__init__.py	0	0	0	100%
accounting/models.py	9	0	0	100%
accounting/tests/__init__.py	0	0	0	100%
accounting/tests/test_urls.py	7	0	0	100%
accounting/tests/test_views.py	20	0	0	100%
accounting/urls.py	3	0	0	100%
accounting/views.py	8	0	0	100%
dashboard/__init__.py	0	0	0	100%
dashboard/admin.py	1	0	0	100%
dashboard/decorators.py	8	1	0	88%
dashboard/forms.py	9	0	0	100%
dashboard/models.py	0	0	0	100%
dashboard/templatetags/__init__.py	0	0	0	100%
dashboard/templatetags/filters.py	9	1	0	89%
dashboard/tests/__init__.py	0	0	0	100%
dashboard/tests/test_forms.py	28	0	0	100%
dashboard/tests/test_urls.py	19	0	0	100%
dashboard/tests/test_views.py	57	0	0	100%
dashboard/urls.py	3	0	0	100%
dashboard/views.py	103	42	0	59%
inventory/__init__.py	0	0	0	100%

inventory/admin.py	10	0	0	100%
inventory/forms.py	22	0	0	100%
inventory/migrations/0001_initial.py	6	0	0	100%
inventory/migrations/__init__.py	0	0	0	100%
inventory/models.py	67	7	0	90%
inventory/tests/__init__.py	0	0	0	100%
inventory/tests/test_forms.py	43	0	0	100%
inventory/tests/test_urls.py	31	0	0	100%
inventory/tests/test_views.py	20	0	0	100%
inventory/urls.py	3	0	0	100%
inventory/views.py	141	89	0	37%
manage.py	12	2	0	83%
manufacturing/__init__.py	0	0	0	100%
manufacturing/admin.py	5	0	0	100%
manufacturing/migrations/0001_initial.py	6	0	0	100%
manufacturing/migrations/__init__.py	0	0	0	100%
manufacturing/models.py	19	2	0	89%
manufacturing/tests/__init__.py	0	0	0	100%
manufacturing/tests/test_urls.py	19	0	0	100%
manufacturing/tests/test_views.py	19	0	0	100%
manufacturing/urls.py	3	0	0	100%
manufacturing/views.py	159	133	0	16%
sales/__init__.py	0	0	0	100%
sales/admin.py	4	0	0	100%
sales/forms.py	27	2	0	93%
sales/migrations/0001_initial.py	6	0	0	100%
sales/migrations/__init__.py	0	0	0	100%
sales/models.py	25	1	0	96%
sales/tests/__init__.py	0	0	0	100%
sales/tests/test_forms.py	15	0	0	100%
sales/tests/test_urls.py	13	4	0	69%

sales/tests/test_views.py	19	0	0	100%
sales/urls.py	3	0	0	100%
sales/views.py	57	37	0	35%
utils/__init__.py	1	0	0	100%
utils/apps.py	5	0	0	100%
utils/migrations/0001_initial.py	7	0	0	100%
utils/migrations/__init__.py	0	0	0	100%
utils/models.py	34	0	0	100%
utils/signals.py	14	0	0	100%
utils/tests/__init__.py	0	0	0	100%
utils/tests/test_email.py	14	0	0	100%
utils/tests/test_models.py	18	0	0	100%
Total	**1174**	**321**	**0**	**73%**

Sprint4 Testing Procedure

3. General Approach

The procedure was to follow the same logic that was adopted in sprint2 for testing. Each application should have 3 tests files; test_urls.py, test_forms.py, and test_views.py. For this sprint, we revisited old files that have a low coverage percentage and if there was an opportunity for improvement it was taken.

4. Coverage Report

For this sprint, our total coverage percentage dropped from 73% to 69%. This is due to the addition of some features. Those features included, having a vendor module where we can create vendors, display their inventory and update it. Also, we have an export button that exports the history of transactions of each module. And many other complementary features. Those features require many lines of code. Moreover, our tests focused mainly on handling the urls entered and directing them to the right function. Also, each form used was tested and made sure that its fields are filled in with the proper information. Finally, the views classes were tested to ensure that when we submit a get or a post request we return the right information.

However, some view classes were noted to have a low coverage and it was a goal for this sprint to test them more. We have achieved that. For example, The views.py in the sales module scored only 35% testing coverage. It was targeted to be under further test. Now, it scored 47%.

Also, it is worth noting that after the addition of many lines of code due to the implementation of the additional features, our test coverage only dropped 4% in total which was an acceptable result by the team.

Below is the coverage report for this sprint:

Coverage report: 69%

Module ↑	statements	missing	excluded	coverage
ERP/__init__.py	0	0	0	100%
ERP/settings.py	32	0	0	100%
ERP/urls.py	4	0	0	100%
accounting/__init__.py	0	0	0	100%
accounting/admin.py	3	0	0	100%
accounting/migrations/0001_initial.py	5	0	0	100%
accounting/migrations/__init__.py	0	0	0	100%
accounting/models.py	9	0	0	100%
accounting/tests/__init__.py	0	0	0	100%
accounting/tests/test_urls.py	7	0	0	100%
accounting/tests/test_views.py	20	0	0	100%
accounting/urls.py	3	0	0	100%
accounting/views.py	45	26	0	42%
dashboard/__init__.py	0	0	0	100%
dashboard/admin.py	1	0	0	100%
dashboard/decorators.py	8	1	0	88%
dashboard/forms.py	9	0	0	100%
dashboard/models.py	0	0	0	100%
dashboard/templatetags/__init__.py	0	0	0	100%
dashboard/templatetags/filters.py	12	2	0	83%
dashboard/tests/__init__.py	0	0	0	100%
dashboard/tests/test_forms.py	28	0	0	100%
dashboard/tests/test_urls.py	22	0	0	100%
dashboard/tests/test_views.py	57	0	0	100%
dashboard/urls.py	3	0	0	100%
dashboard/views.py	185	103	0	44%
inventory/__init__.py	0	0	0	100%

inventory/admin.py	10	0	0	100%
inventory/forms.py	22	0	0	100%
inventory/migrations/0001_initial.py	6	0	0	100%
inventory/migrations/__init__.py	0	0	0	100%
inventory/models.py	67	5	0	93%
inventory/tests/__init__.py	0	0	0	100%
inventory/tests/test_forms.py	43	0	0	100%
inventory/tests/test_urls.py	31	0	0	100%
inventory/tests/test_views.py	20	0	0	100%
inventory/urls.py	3	0	0	100%
inventory/views.py	167	107	0	36%
manage.py	12	2	0	83%
manufacturing/__init__.py	0	0	0	100%
manufacturing/admin.py	5	0	0	100%
manufacturing/migrations/0001_initial.py	6	0	0	100%
manufacturing/migrations/__init__.py	0	0	0	100%
manufacturing/models.py	19	2	0	89%
manufacturing/tests/__init__.py	0	0	0	100%
manufacturing/tests/test_urls.py	19	0	0	100%
manufacturing/tests/test_views.py	19	0	0	100%
manufacturing/urls.py	3	0	0	100%
manufacturing/views.py	179	145	0	19%
notifications/__init__.py	1	0	0	100%
notifications/apps.py	5	0	0	100%
notifications/forms.py	23	11	0	52%
notifications/migrations/0001_initial.py	8	0	0	100%
notifications/migrations/__init__.py	0	0	0	100%
notifications/models.py	90	27	0	70%
notifications/signals.py	34	14	0	59%
notifications/templatetags/__init__.py	0	0	0	100%
notifications/templatetags/notifications_tags.py	58	26	0	55%

notifications/tests/__init__.py	0	0	0	100%
notifications/tests/test_email.py	14	0	0	100%
notifications/tests/test_models.py	18	0	0	100%
notifications/urls.py	3	0	0	100%
notifications/views.py	31	14	0	55%
sales/__init__.py	0	0	0	100%
sales/admin.py	5	0	0	100%
sales/forms.py	27	2	0	93%
sales/migrations/0001_initial.py	6	0	0	100%
sales/migrations/__init__.py	0	0	0	100%
sales/models.py	32	1	0	97%
sales/tests/__init__.py	0	0	0	100%
sales/tests/test_forms.py	26	0	0	100%
sales/tests/test_urls.py	13	4	0	69%
sales/tests/test_views.py	44	0	0	100%
sales/urls.py	3	0	0	100%
sales/views.py	127	67	0	47%
vendors/__init__.py	0	0	0	100%
vendors/admin.py	1	0	0	100%
vendors/forms.py	21	0	0	100%
vendors/models.py	1	0	0	100%
vendors/tests/__init__.py	0	0	0	100%
vendors/tests/test_forms.py	29	0	0	100%
vendors/tests/test_urls.py	19	0	0	100%
vendors/tests/test_views.py	55	0	0	100%
vendors/urls.py	3	0	0	100%
vendors/views.py	82	26	0	68%
Total	**1863**	**585**	**0**	**69%**