# 4. Реализация

В данном курсовом проекте кроме реализации самого проекта выполнена реализация ряда дополнительных, вспомогательных процессов, таких как тестирование и непрерывная интеграция.

## Тестирование

Тестирование в проекте представлено двумя видами, описанными далее.

### Модульное тестирование

Модульное тестирование проводилось для ряда приложений, представленных в проекте, но так как не все из этих приложений имеют функции, пригодные для проверки, то проверка осуществлялась не только для уникальных функций, но и для встроенных.

Для написания тестов класс, осуществляющий проверку, должен наследоваться от TestCase. TestCase – тестовый сценарий (набор проверяемых условий, переменных, состояний системы или режимов). Обычно является логически неделимым и может содержать одну или более проверок (asserts). [1] Также все методы классов-тестов должны начинаться с выражения test\_.

Далее представлен исходный код методов и описано, какую проверку они осуществляют.

#### Тесты Place

Для приложения Place представлены простейшие проверки модели – недопустимость создания сущности без обязательных полей. Эти проверки разбиты на два логических модуля – один проверяет недопустимость создания сущности вообще без атрибутов, другой без атрибута пользователя.

Исходный код этих тестов представлен далее.

class PlaceTest(TestCase):

def test\_null\_place(self):

with self.assertRaises(IntegrityError):

p = Place()

p.save()

def test\_place\_without\_user(self):

with self.assertRaises(IntegrityError):

p = Place.objects.create(

name='test',

)

#### Тесты Category

Для приложения Category представлено два теста, один из которых проверяет, что атрибут name созданной сущности возвращается как название сущности. Второй же тест проверяет валидацию модели, то есть недопустимость создания категорий с одинаковыми именами и одинаковыми родительскими полями (по умолчанию они пустые).

Исходный код этих тестов представлен далее.

class CategoryTest(TestCase):

def test\_trycategoryname(self):

user = get\_user\_model().objects.create(

username='test',

)

cat = Category(name='TestCategory', parent=None, user=user)

self.assertEqual(cat.\_\_str\_\_(), 'TestCategory')

def test\_cleancategory(self):

user = get\_user\_model().objects.create(

username='test',

)

Category.objects.create(name='test', user=user)

cat2 = Category(name='test', user=user)

self.assertRaisesMessage(ValidationError, 'Name must be unique', cat2.clean)

#### Тесты Transaction

Для приложения Transaction представлен тест, проверяющий валидацию модели, которая запрещает создавать транзакцию с одинаковыми значениями полей account\_from и account\_to, то есть «счёт откуда» и «счёт куда».

Исходный код этого теста представлен далее.

class TransactionTest(TestCase):

def test\_double\_amount(self):

user = get\_user\_model().objects.create(

username='test',

)

# Transaction.objects.create(account\_from='test', account\_to='test', user=user)

acc = Account(name='qwe', user=user)

acc.save()

trans2 = Transaction(account\_from=acc, account\_to=acc, user=user)

self.assertRaisesMessage(ValidationError, 'Счета должны быть разные', trans2.clean)

#### Тесты Account

Для приложения Account тест представляет проверку собственного метода balance, который подсчитывает текущий баланс для счёта.

Исходный код этого теста представлен далее.

class AccountTest(TestCase):

def test\_balance(self):

amount = Decimal(1.23)

user = get\_user\_model().objects.create(

username='test',

)

account = Account.objects.create(

name='test',

user=user,

)

Transaction.objects.create(

account\_to=account,

amount=amount,

)

self.assertAlmostEqual(account.balance(), amount, 2)

Все описанные тесты проходят проверку, что подтверждает правильность как написания тестов, так и ожидаемую реакцию функций на производимые в тестах проверки.

Правильность подтверждается прохождением тестов в среде PyCharm. Результат запуска тестов представлен на рисунке 1.

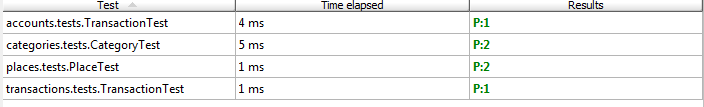


Рисунок 1 – Результат тестов

## Непрерывная интеграция

Для проекта также настроена непрерывная интеграция. Непрерывная интеграция (англ. Continuous Integration) — это практика разработки программного обеспечения, которая заключается в выполнении частых автоматизированных сборок проекта для скорейшего выявления и решения интеграционных проблем. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. Она может непредсказуемо задержать окончание работ. Переход к непрерывной интеграции позволяет снизить трудоёмкость интеграции и сделать её более предсказуемой за счет наиболее раннего обнаружения и устранения ошибок и противоречий. [2]

Для данного курсового проекта используется Travis CI. Travis CI — распределённый веб-сервис для сборки и тестирования программного обеспечения, использующего GitHub в качестве хостинга исходного кода. Программная составляющая сервиса также располагается на GitHub’е, однако разработчики не рекомендуют использовать её в закрытых проектах. [3]

Процесс настройки довольно прост, и представляет собой последовательность четырёх шагов, которые описаны далее.

Для использования сервиса необходимо зайти через свой GitHub аккаунт и дать разрешение на доступ к ряду параметров аккаунта, основным разрешением является доступ к webhook [4]. Для входа достаточно быть авторизированных на хостинге GitHub и нажать Sign in with GitHub, представленную на рисунке 2.



Рисунок 2 – Вход в Travis CI

После этого необходимо перейти в свой аккаунт и активировать построение репозиториев нажатием переключателя. Результат этого процесса представлен на рисунке 3.



Рисунок 3 – Включение построения репозитория

Далее следует создать файл .travis.yml, который содержит описание репозитория – на каком языке написан исходный код, на какой версии, и дополнительные параметры, такие как список необходимых пакетов для построения проекта. Файл для данного курсового проекта представлен далее.

language: python

python:

- "3.2"

- "3.3"

# command to install dependencies

install: "pip install -r requirements.txt"

script:

- "flake8 ."

- "python manage.py test"

В этом файле стоит отметить два момента, описанных в script. Первая строка flake8 ., запускает проверку всего проекта на «чистоту» кода, вторая строка запускает тесты.

Для запуска построения необходимо сделать фиксацию изменений в репозитории.

После того, как Travis CI осуществит построение репозитория, то этому построению присвоится статус «Failed», если были допущены ошибки и «Passed», если ошибок нет.

## Развёртывание

Диаграмма развёртывания представляет собой классическую схему для веб-приложения. На стороне сервера установленная операционная система Linux с веб-сервером Apache. На стороне клиента может быть любой браузер и любая операционная система, для примера взят Firefox и Windows.

Также для успешного запуска приложения в системе, наряду с веб-сервером, должен быть установлен Python и Django.

Выбор веб-сервера обоснован тем, что Django проектировался для работы под управлением Apache с модулем mod python и с использованием PostgreSQL в качестве базы данных. [5]

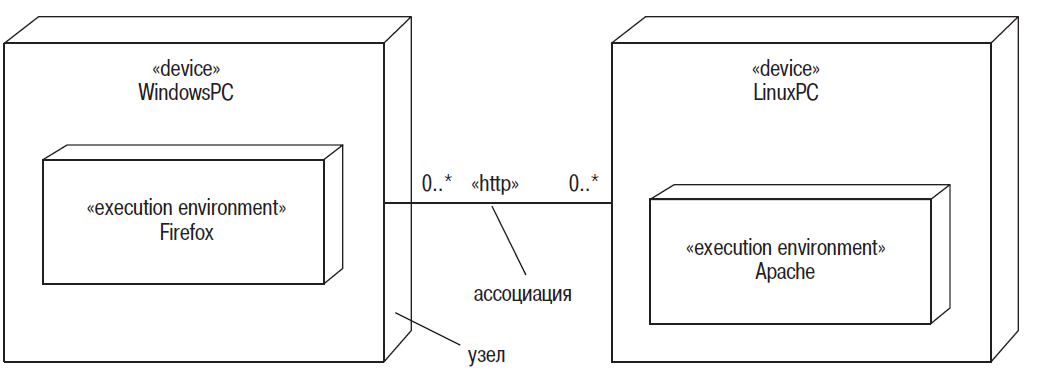


Рисунок 4 – Диаграмма развёртывания

# Список литературы

1. <http://gahcep.github.io/blog/2013/02/10/qa-in-python-unittest/>
2. <https://ru.wikipedia.org/wiki/Непрерывная_интеграция>
3. <https://ru.wikipedia.org/wiki/Travis_CI>
4. <http://en.wikipedia.org/wiki/Webhook>
5. <https://ru.wikipedia.org/wiki/Django>