Advanced Techniques and Tools for Software Development

ToDDo Application

Implementation and Analysis

Author: Davide Galassi
Supervisor: Prof. Lorenzo Bettini
Last Update: 17-02-2018
Version: 1.0.1

Table of Contents

Keywords

Follows a table with some keywords that are present in this document.

Keyword	Description
PEP8	Style guide for Python code document
Pip	*De-facto* official Python package manager
TDD	Test Driven Development
URL	Uniform Resource Location
VCS	Version Control System
Mock	A fake class/method instance that is used to substitute an external dependency during an test.

Tools Overview

- **Programming Language :** Python-v3
- **Dependencies Management :** Pip-v3
- **Unit Tests Tools :** Python built-in `unittest` library
- **Code Coverage:** “coverage” tool with results published to `coveralls.io` cloud service
- **Functional Tests Tools :**
  - selenium: browser driver
  - xvfb : virtual frame buffer
- **Web Framework :** Django-v2.0
- **Database :** SQLite-v3
- **VCS :** Git and GitHub for sources sharing
- **Continuous Integration and Testing:** Travis-CI cloud service
- **Static analysis:** SonarQube with results published to `sonarcloud.io` cloud service
- **Environment Reproduction:** Docker
  - docker-compose: containers orchestration
  - Nginx : webserver and proxy
  - Gunicorn : wsgi django server runner

Project Links

GitHub: https://github.com/davxy/toddo

Trevis-CI: https://travis-ci.org/davxy/toddo

Coveralls: https://coveralls.io/github/davxy/toddo

SonarCloud: https://sonarcloud.io/organizations/davxy-github

DockerHub: https://hub.docker.com/r/davxy

Conventions

Shell commands

```
$ command
```

Python 3

Python3 is not 100% syntactically compatible with the predecessors. In this document `python` and `pip` commands are used as synonyms for `python3` and `pip3`, respectively. This assumption is made even if the two, in some distributions, are symlinked to Python2 binaries.

Comments

As per PEP8, documentation comments (*docstrings*) should use the `'''` or the `"""` form. They are usually used to describe a module, function, class or method purpose and how to use it.

```
def factorial(n):
       '''
       Computes factorial of value n using recursion.
       Args:
          n : the value
       Returns:
          The factorial of n
       '''
       if n < 0: raise ValueError('negative input')
       else if n < 2: return 1
       else: return n * factorial(n-1)
```

Code comments, that is comments that explains some details about a snip of code, should use the "hash" form.

```
# The user checks his email and finds a message
# (this exploits the dummy email backend)
email = mail.outbox[0]
```

Document content

The Project

This section describes the *ToDDo* application from both a design and an implementation point of view. This part also shows how to use the application user interface and the user authentication procedure.

Development Tools

The core of this document. In this long section we'll examine some of the more advanced tools and techniques used in software development. Test driven development, mocking, continuous integration, code coverage, static analysis with the support of widespread tools like Git and Github, Travis-CI, Doker and SonarQube.

Known Issues

Problems that are known and for which a solution has not been found or that requires an investigation and techniques that are out of the scope of this document.

Additional Notes

Supplementary notes that are worth to be read.

Appendix 1 - Django Overview

A very high level description of the Django framework architecture. This paragraph is not meant to be a complete tutorial of the software, is just an explanation of the core components that would be useful to better follow some parts of our discussion if you've never used the framework.

Appendix 2 - Quickstart

Essential steps that are required to quickly get the application up and running.

The Project

Introduction

ToDDo is a simple todo-list web application implemented using some of the “state of the art” development techniques and tools available today.

The application is trivial “*by-design*” and acts more as a working medium used to illustrate how the various techniques and tools we’ll see fits together in a test driven development process. As such, it should be used as a didactical tool and its not meant for production.

Why *ToDDo*

ToDDo is the vector used to target, above the others, Test Driven Development methodologies. As such, the name just derives from the TDD acronym with a couple of extra lowercase “o”s to resemble the intended usage of the application, i.e. a to-do list.

User Interface

At the glance, the interface appears very minimal and lean, the application user is taken directly to the point.

There are two kind of users:

- **Guest users**: occasional visitors. Their lists are grouped together and are of public domain.
- **Registered users**: visitors that have received their authentication token via mail. Registered users can create private and persistent lists.

When a guest user visits the home page he can start a bare new list, open an existing one to examine existing tasks or append new ones, or he can decide to register to the application by inserting an email address.

If the user decides to register, an authentication token is generated and sent, embedded in a URL, to the user provided email. To authenticate himself within the system the user just have to visit such a link[1].

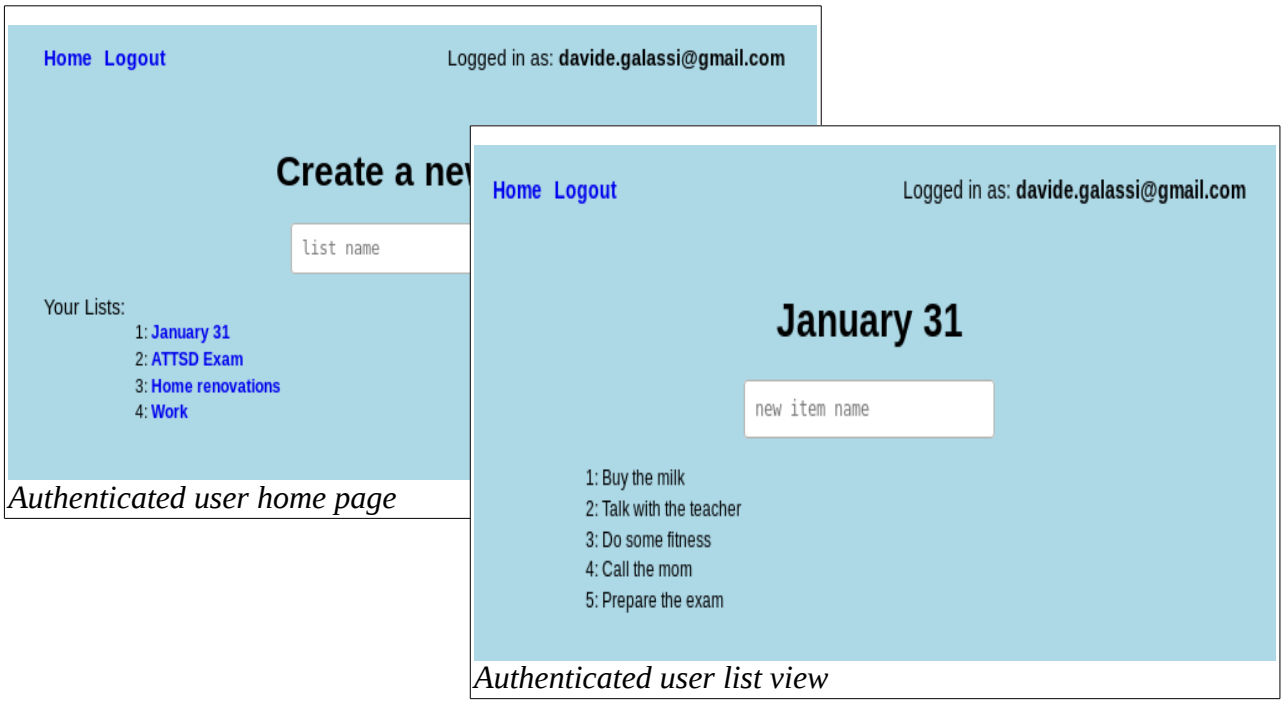
1. We want to highlight that this authentication method is not meant to be secure but is a feature that will be useful when we’ll talk about dependency mocking.

Anonymous user home page

Once that the email is sent, and thus the authentication token has been created, the corresponding user is not immediatelly created within the system database. It's creation is postponed to the first visit of the authentication link.

When the token is created, its generation time-stamp is also saved. The time-stamp allows an administrator, or an automatic program, to monitor the database and eventually remove tokens that are not activated after a given period of time.

To the logged user the interface appears similar, but the registration form is replaced with its identity and a "Logout" link is visible. This link can be clicked whenever the user wish to terminate its session.



Authenticated user home page

Authenticated user list view

Implementation

Modules

The application sources are divided in three macro modules:

- **lists**: management of lists and tasks;
- **accounts**: management of users and authentication subsystem;
- **toddo**: contains application-wide settings and variables.

Application components

Django does most of the management "*dirty work*" under the hood, our job has been to correctly implement the missing pieces of the puzzle that together allows the framework do something useful. In particular, the main activities consisted in:

- Create the required application models classes;
- Add valid routes in the URL files to redirect http requests to our views;
- Implement the views routines to create, after some logic and sanity checks, the http responses.
- Create the templates used by the views to dynamically create the responses html payload.

For more details about the *Django* framework architecture components please refer to the Appendix.1. What follows are details of how we've implemented such components.

Models

The "accounts" module defines the `Token` and the `User` classes. The "lists" module defines the `List` and the `Item` classes.

A `Token` represents the unique uid generated to be sent via email to the user. The model is composed by an `email`, a `creation` date-time and a `uid`.

A `User` represents a registered user and it is instanced the first time the authentication link is visited. It has only the `email` field. This model is also used by the built-in *Django* authentication subsystem, thus to be compliant with the expected interface, some additional fields were added. These fields are internally used by *Django* and are not important for the scope of this discussion.

A `List` represents a to-do list and is composed by its `name` and the `owner`, a reference to a `User` model instance.

An `Item` represents a to-do list element and is composed by the a `text` field and a reference to a `List` instance.

Routes

Django takes the routes from a list in the the main module URL file, `toddo/url.py`. We've extended the default routes list by including the ones defined by our application modules, `accounts/urls.py` and `lists/urls.py`. Every valid url will allow the *Django* "router" to univocally identify and invoke one of the view routines we've implemented.

Views

Is in the views that most of our code logic is implemented. This part is also the one that directly interact with the application "model" for CRUD operations.

In the lists views file, `lists/views.py`, contains the code to enumerate, by name, all the lists when the home page is visited and the code to enumerate a list entries (tasks) when a particular list is clicked. The correct list is selected within the view using a function parameter. This parameter is the list numerical identifier and is automatically extracted by the *Django* "router" from the URL (e.g. http://localhost:8080/lists/**3**/) thanks to a "tag" we've inserted in the URL file.

In the accounts views file, `accounts/views.py`, resides the code to log-in and log-out a registered user. In contrast to the logout view, which url link is visible in the user interface, the login view is not directly invoked. The login view is implicitly invoked when the user visits the authentication url that the application sent by email.

The accounts views also contains the code to send the authentication email. This view is invoked by the email input form and exploits the library "send_mail" function. Note that the new token is created only if the user email is not already registered and saved only if the email is sent correctly.

Templates

Templates are *pseudo* HTML documents that allows to embed some *Django* custom scripting code and variables (a bit like how `php` is embedded in HTML). In addition templates inheritance is allowed, that means that a template can be easily encapsulated into another. The "base" template can also define some placeholders that the sub-templates can overwrite with specialized values.

The *ToDDo* application uses three templates: a base one, *base.html*, from which two specialized ones inherits: *home.html* and *lists.html*, used to render the home page and the single lists, respectively.

User Authentication

The Django framework allow us to use our own user model with the authentication subsystem. We've created a User model that contains only the email field, thus the application does not follow the "classic" user name and password pattern.

To inform Django of the model to be used for authentication, we must specify it in the "`toddo/settings.py`" file via the AUTH_USER_MODEL variable.

Very roughly and without entering too much in the details, the authentication backend works by setting a cookie in the user browser. When a user http request contains this cookie then the Django authentication engine is able to recognize from what user the request comes from and if it is logged-in.

The correct user is then set into the current HttpRequest class instance before is passed to the view for further processing.

Development Tools

GIT Version Control System

As a VCS the ubiquitous *git* tool has been used.

For the *power-user*, the tool manifests its full potential from the command line however, for convenience, very often its usage is accompained by a GUI tool, especially for branch and `diff` analysis.

Plenty free GUIs exists and the choice is a very subjective matter. We've preferred the *Gnome* friendly `gitg` that integrates, in one single tool, the functionalities of two other widespread tools: `gitk`, for history analysis, and `git-gui`, for staging, commit, and others useful operations.

The main *ToDDo* repository is hosted by *GitHub*, the first choice for open source developers nowadays.

Branching model

Even if branching and merging can be an arbitrary practice, meaning that branches can be created, merged and pruned as and when the developer likes, to avoid confusion several more-or-less strict models were proposed by the developers community.

We've decided to follow the "*GitFlow*" model proposed by *Vincent Driessen* (http://nvie.com/posts/a-successful-git-branching-model). The model, or its variations, is the most adopted one because it is fairly strict and generally suits well the needs of (almost) every software projects:

- have a stable, production grade, branch;
- a less stable development branch;
- work on new features in a dedicated branch, without break other branches working code;
- immediatelly report urgent bug-fix to the production branch.

Thus, in general, the branches that are found in the *ToDDo* project history are:

- `master` : stable release with tagged versions. Only hotfix and develop merges are allowed here.
- `develop` : development branch from were the new features branches diverge and converge.
- `feat/name` : new features started from the develop branch.
- `hotfix`: branch started from master to fix an important issue that can't wait for the next official release. The branch changes are immediately merged into the master and develop.

The only “violations” to the *Drissen* model is that the “release” branch is not used. Such a branch is considered useful for critical, production grade, projects that are maintained in a freezed state (release candidates) for a variable period of time before being released and thus merged into the master. In our context that strictness is not required.

Branch merging

Merging can be managed in two ways: directly on the local machine or remotelly by using the *GitHub* fork and pull-request model.

The second approach has been choosen for two reasons: allows discussions of the implemented modifications and features with other developers, even if they are not directly involved with the project, and gives an eagle-eye overview of the modifications as a whole, easily and to everyone.

Such an overview feature is also provided by local applications such as gitg, gitk, git-cola, etc., but a local tool doesn’t have a built-in quick qay to share its perspective with the other team members or the community. Without tools like *GitHub*, people that want to analyze our modifications is forced to pull the project changes to their local machine.

Test Driven Development

All the functionalities were implemented by rigorously following TDD techniques.

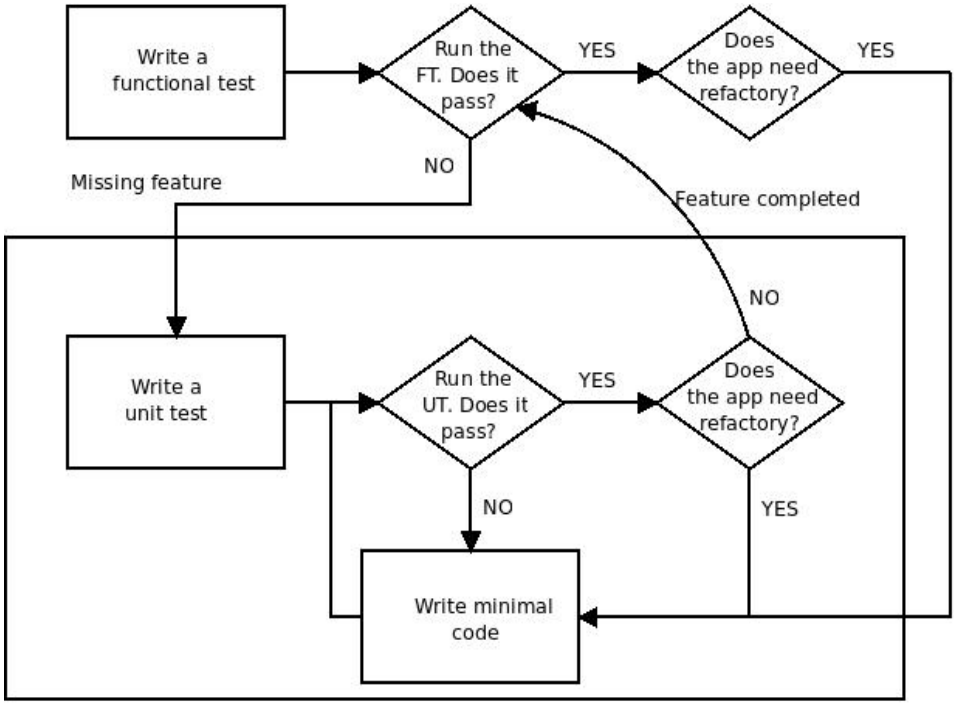
- Project stakeholders requirements were collected. Here there are no real stakeholders, so we’ve immagined what a to-do list user may expect from such an application.
- These requirements were translated into functional tests. At this stage, the functional tests are written to fail! Their main purpose is to collect the expectations and behaviours of a potential application user.
- To let the functional test succeed, the single pieces that constitutes the application should be implemented step by step, constructed incrementally. The two macro pieces of the application that requires attention were the object model and the views.
- Features were added incrementally to the models and the views by first writing a unit test that exercise the required feature. Expecting an initial failure, since the feature is not present yet.
- Finally, test after test, the functional test should pass. When we reach such a point we can assert that the user requirements were satisfied, at least with regard to that functional test.

All this process can be iterated several times, in particular there are two types of iterations:

- *inner*: iteration through a unit test;
- *outer*: iteration through a functional test.

The outer iteration usually tries to satisfy a massive requirement from the client. A change in the functional test may require several changes to the single application pieces and thus several inner iterations. Even a small change to the initial requirements can influence the code logic in a way that takes several, previous successful, unit tests to fail.

Follows a picture that tries to synthesize the concepts and the logical paths of the followed TDD process.



Unit tests

Unit tests were developed by using the *Django* `TestCase` class, an indirect descendant of the built-in `unittest` standard Python class, and as such the test runtime will automatically invoke:

- the setUpClass and tearDownClass functions, before and after all the tests in a test class are executed;

- the setUp and tearDown methods, before and after each test method within a test class.

The `TestCase` class, with its `setUp`/`tearDown` methods provides a priceless fixture: it creates an in-memory database that can be used by our models during the tests. The data that we've stored within the database is flushed in the tearDown mehtod, so that a test is not influenced by the previously executed one.

Having this test database, for us, means that we are not forced to mock every call that the *Django* object model can made against the database backend during the tests execution. We can safely assume that the database works as expected. For demostration purposes we'll mock something else.

Functional tests

All the functional tests inherit from the *Django* `LiveServerTestCase` class. When this class `setUp` function is invoked, it offers another fundamental fixture: it automatically starts a test server listening on a random port of the local host.

In web development a functional test is mainly about simulation of the user possible interactions with our application through a web browser. This sort of interaction can be easily emulated by using the `selenium` package, an important library that allows to perform some of the common human actions: visit pages, click buttons, fill forms, etc.

To work properly, selenium requires specific browsers drivers (e.g. *gekodriver* for Firefox) the `webdriver_manager` package were initially used to download the required drivers from the application functional test startup routines.
But after a while we've decided to remove its direct usage and manually download the drivers when required. There are a couple of practical motivations for this choice:

- The library requires an Internet connection, and it tries to connects several times. Possibly slowing down the test routine. This is done even if the driver is already present in the download directory (probably because it checks for newer versions). This can let the test fails when we are not connected to the Internet.
- When it is executed on the remote Travis-CI machine very often we've received the "download limit reached" error for the drivers downloaded from *GitHub* like the Firefox one.

The second issue can be solved by adding a *GitHub* auth token to Travis (encrypted!), in this case the limit value is increased from 50 to 5000. Anyway, to not loose precious time we've decided to manually download the drivers and concentrate to more important aspects.

Note that our choice does not question the utility of the *webdriver-manager* library, indeed it is still used in a stand alone python application (`tools/getdriver.py`).

Send-mail test backend

The test classes `setUp` method also provides another fixture very usefull for our application testing. The *Django* `send_mail` is replaced with a dummy backend that has an “outbox” list. When the dummy send_mail is invoked, in place to contact the SMTP server specified in the configuration, a new structure with sender, subject, body and recipient is created and pushed into the outbox list.

Our “accounts” functional test uses this fixture to check that indeed the outstanding email contains what we expect. Thus, we can proceed by assuming that the “good” `send_mail` works as expected and use this *Django* provided mock (note the analogies with our hand written mock example presented later).

Environment Parameters

The following environment variables, when defined, alters the functional tests behaviour.

`SHOW_DISPLAY = 1`
: Avoid use the virtual frame buffer and open the application gui in the current display.

`SKIP_FIREFOX = 1`
: If defined then the functional tests for Firefox are skipped.

`SKIP_CHROME = 1`
: If defined then the functional tests for Chrome are skipped.

To be effective a parameter should be exported in the test application environment before its execution. For example:

```
$ export SHOW_DISPLAY=1; python3 manage.py test -v2
```

Mocking

In python, dependency mocking can be done "*by-hand*" or by using the built-in mock library (`unittest.mock`). For the features offered, and to be compliant with the Python best practices, our project relies on the built-in library.

The former method is inserted in the project, and explained here, just for didactical purposes.

Manual mocking

A demonstration of manual mocking (aka *monkey mocking*) is provided for the accounts unit tests. Follows the parts of the unit test class that shows how to manually patch an object method.

accounts/tests/test_send_login_email_view_monkey.py

```
class SendLoginEmailViewMonkeyTest(TestCase):

    def send_mail_mock(self, subject, body, from_email, to_list):
        """mock used to patch the send_mail"""
        self.called = True
        self.subject = subject
        self.body = body
        self.from_email = from_email
        self.to_list = to_list
        return 1


    def setUp(self):
        """Set up the object by patching the “good” send_mail"""
        super().setUp()
        # Save the original send_mail method
        self.original_send_mail = accounts.views.send_mail
        # Swap out the real accounts.views.send_mail with our fake version.
        accounts.views.send_mail = self.send_mail_mock


    def tearDown(self):
        # Restore the original send_mail method
        accounts.views.send_mail = self.original_send_mail
        super().tearDown()


    def test_sends_mail_to_address_from_post(self):
        # This resolves to a view that calls the send_mail_mock method
        self.client.post('/accounts/send_login_email', data={'email': TEST_EMAIL})

        # check that the mock has been invoked
```

```
        self.assertEqual(self.called, True)
        # check that the mock has been called with the expected arguments
        self.assertEqual(STRINGS['mail_subject'], self.subject)
        self.assertIn(STRINGS['mail_body'], self.body)
        self.assertEqual(STRINGS['mail_origin'], self.from_email)
        self.assertEqual([TEST_EMAIL], self.to_list)
```

Python Mock library

The built-in library does exactly what we've done with manual mocking, but more easily and transparently.

To patch a method/function within a test method context, we just need to "*decorate*" the test method/function with the `patch` decorator.

The following test does the same things we've done with the previous, more verbose, manual patching.

accounts/tests/test_send_login_email_view.py

```
class SendLoginEmailViewTest(TestCase):

    # The send_mail is mocked, when invoked returns 1.
    # The mock object (mock_send_mail) that has been instanced by the lib to
    # patch the real send_mail is passed to the test method as an additional
    # argument.
    @patch('accounts.views.send_mail', return_value=1)
    def test_sends_mail_to_address_from_post(self, mock_send_mail):
        # This resolves to a view that calls the mocked send_mail method
        self.client.post('/accounts/send_login_email', data={'email': TEST_EMAIL})

        # Get all the arguments that were used to invoke the send_mail mock.
        # The library stores all those arguments in the "call_args" mock parameter.
        (subject, body, from_email, to_list), _kwargs = mock_send_mail.call_args
        # 'called' is automatically set to True when the mock is invoked
        self.assertEqual(mock_send_mail.called, True)
        # Check that the mock hash been called with the expected arguments
        self.assertEqual(STRINGS['mail_subject'], subject)
        self.assertIn(STRINGS['mail_body'], body)
        self.assertEqual(STRINGS['mail_origin'], from_email)
        self.assertEqual([TEST_EMAIL], to_list)
```

Note that the original method is automatically saved and restored by the library before and after the test, there's no need to do it manually in the `setUp` and `tearDown` methods.

In the monkey patching version, if we forgot to restore the original `send_mail` method, then this will remain patched during all the test session, and may influence later tests that may even reside in other test classes and that don't want to use our patched version (i.e. in the `test_login` functional test).

Continuous Integration

Travis-CI cloud service has been used to provide continuous integration.

To let *GitHub* use the Travis service we must insert a "`.travis.yml`" file in the root of our project. This file describes the actions that the Travis machine should perform with our cloned repository.

In our case:

- download the browser drivers required by selenium;
- install all the dependencies with *Pip;*
- execute all the unit and functional tests with code coverage reporting;
- statically analyze our project using Sonar scanner.

Functional tests are remotelly executed thanks to the `xvfb` (virtual frame buffer) Linux package[2].

The selenium browser-drivers and the sonar cache files are required to be cached by Travis to prevent continuous downloads.

Follows the Travis configuration file used by the project

`.travis.yml`

```
language: python

python:
  - "3.6"

sudo: required
addons:
  chrome: stable
  sonarcloud:
    branches:
      - master
      - develop
    organization: "davxy-github"
    token:
      secure: "very-long-encrypted-string"
```

2 We set up the display and start the `xvfb` service directly from Pytohn using the `PyVirtualDisplay` library.

```
  before_install:
    - mkdir -p $HOME/drivers && cd $HOME/drivers
    - wget -nc
https://github.com/mozilla/geckodriver/releases/download/v0.19.1/geckodriver-
v0.19.1-linux64.tar.gz
    - tar -xzf geckodriver-v0.19.1-linux64.tar.gz
    - wget -nc
http://chromedriver.storage.googleapis.com/2.30/chromedriver_linux64.zip
    - unzip -o chromedriver_linux64.zip
    - cd -

  env:
    - DJANGO_VERSION=2.0 PATH=$PATH:$HOME/drivers

  install:
    - pip install -q Django==$DJANGO_VERSION selenium PyVirtualDisplay
coverage python-coveralls pylint

  script:
    - coverage run manage.py test -v2
    - coverage xml -i
    - sonar-scanner

  after_success:
    - coveralls

  cache:
    directories:
      - $HOME/drivers
      - $HOME/.sonar/cache
```

Code coverage

Coverage analysis is provided by the python `coverage` package.

```
$ pip install coverage
```

To generate coverage information, a program is started using the "`coverage run`" command instead of directly invoke the Python interpreter. In our case

```
$ coverage run manage.py test -v2
```

To check for code coverage of our sources only, instead of the whole Django and tests code a configuration file, "`.coverage.rc`", should be placed in the directory from where the tool is started, in our case the project root directory. This is the content of the file stored in our project:

```
[run]
include =
      lists/views.py
      lists/models.py
      accounts/views.py
      accounts/models.py
      accounts/authentication.py
```

The data can be examinated locally (e.g. html, csv or console tables) or sent to the `coveralls.io` cloud service for web publication, in this case an additional, `python-coveralls`, tool should be used.

```
$ pip install python coveralls
```

The information is sent to the cloud service by just issuing the `coveralls` command from the directory containing the coverage analysis results.

```
$ coveralls
```

In our application, coverage reports are always sent to `coveralls.io` service when tests are executed via *Trevis-CI*.

Example of coverage results displayed locally after that the coverage data has been harvested:

```
$ coverage report
Name                          Stmts   Miss  Cover
-----------------------------------------------
accounts/authentication.py       15      0   100%
accounts/models.py               12      0   100%
accounts/views.py                45      0   100%
lists/models.py                   8      0   100%
lists/views.py                   56      0   100%
-----------------------------------------------
TOTAL                           136      0   100%
```

Docker

The docker tool is used to uniformly reproduce working environments in a quick and simple manner. Both for development and deployment.

A docker-hub account has been set up to contain three *Debian* machines:

- one to run the server code directly from the *Django* built-in server (`davxy/django`);
- one like the previous option but by passing through an *Nginx* proxy (`davxy/nginx-django`);
- one to run the server code via *Gunicorn* and passing through an *Nginx* proxy (`davxy/nginx-gunicorn`).

The machines doesn't directly embed any *Django* project but are built to start whatever project the user wants to run. The project to run is picked from a host directory mounted in the container as a virtual volume.

The Dockerfiles to build the machines, together with the scritpt to practically start them, are maintained within the *GitHub* repository of the *ToDDo* project, under the `tools/dockerfiles` directory. Pre-built images images were published to the *docker-hub* community site.

Containers Orchestration

To manage the containers we've adopted two methods:

- A rough Posix shell script that launch the containers by encapsulating the commands that should be given from the command line (with some options and default values).
- The widespread `docker-compose` tool, a tool that allows to start and stop easily several containers, manage private networks and dependencies using a simple `yml` file.

More information about the custom script or a demostration of how to use `docker-compose` please refer to the Appendix 2 – Quickstart.

SonarQube

SonarQube is a client/server application meant to support code quality analysis.

Server

The server can be run via *Docker* using one of the *SonarQube* official images. The latest image already comes with the Python plugin installed. The container can be started as follows:

```
docker run -d -p 9000:9000 sonarqube
```

From now on the server service can be acessed from the local host port 9000.

If you want to keep track of your progress history, it is important to preserve the *SonarQube* data between container runs. Thus, we need to attach the directories where the server stores the project data and database files as external volumes. How to do this is described in great detail in the Lorenzo Bettini's *SonarQube* tutorial.

Client

The official client application is available at the following address:

```
https://sonarsource.bintray.com/Distribution/sonar-scanner-cli/
```

The client is a Java application and because our project does not use Java at all, for a potential contributor installing all the java environment in the local machine just to publish the *SonarQube* information would be an overkill.

As a solution, we've decided to run the client in a Docker container as well. In docker-hub, there is someone that already thought about it for us and provides a pre-built image on `docker-hub` with the *SonarQube* scanner tool already installed. We exploit its work: `zaquestion/docker-sonarqube-scanner`.

From our project root, we start the container as follows:

```
docker run -ti -v $PWD:/root/src zaquestion/sonarqube-scanner
```

Once started, the container runs the following command:

```
sonar-scanner -D sonar.projectBaseDir=./src
```

To work properly, in the directory from where the scanner container is started, there should be a file named `sonar-project.properties` containing the scanner configuration directives. Follows the one that we've used when running against our local server

sonar-project.properties

```
# Required metadata
sonar.projectKey=toddo
sonar.projectName=ToDDo
sonar.projectVersion=1.0

# Comma-separated paths to directories with sources (required)
sonar.sources=lists,accounts

# Files to analyze, skip auto generated Django files
sonar.inclusions=lists/views.py,lists/models.py,accounts/views.py,
                  accounts/models.py,accounts/authentication.py

# Language
sonar.language=py

# Encoding of the source files
sonar.sourceEncoding=UTF-8

# Server URL
sonar.host.url=http://172.17.0.2:9000
```

Because the scanner is started as a docker container as the server URL we need to use the address that has been assigned to the server container within the docker private network. Assuming an already running server, its address can be found with the following command:

```
$ docker network inspect bridge
```

To send `coverage` tool report results to the server, first generate an xml version of the coverage results

```
$ coverage xml -i
```

then add the following line to the sonarqube configuration file.

```
sendsonar.python.coverage.reportPath=coverage.xml
```

Publishing

After we've taken some confidence with the tool, we've decided to switch to the cloud `sonarcloud.io` service to publish our results.

The publication requires to authenticate the client within the sonarcloud platform. We've followed the best practice to put the authentication token, generated from sonarcloud, encrypted in the *Travis-CI* file.

More on this argument can be found in the Travis website, in the sonarcloud section:

```
https://docs.travis-ci.com/user/sonarcloud/
```

In the Travis file, other than the sonarcloud section with the token, branches and organization fields, we've added some instructions to be executed by Travis. Our script section looks like the following:

```
script:
    - coverage run manage.py test -v2
    - coverage xml -i
    - sonar-scanner
```

Rules list

By default, for Python, *SonarQube* uses a rules list named "*Sonar Way*". This list is very minimal since is composed by only 38 rules of the more than 200 available. We've decided to be more pedantic and to create our own rules set, named "*ToDDo way"*, composed by all but the deprecated rules.

During the way, to prevent some "code smells", we've got to turn off or ignore a couple of rules:

- "*Class with too few members*" : we've obtained this warning for the model classes. Indeed such classes have just a couple of member. But that's how the model should be.
- "*Access to class nonexistent member*" : obtained for the model classes. Effectivelly we access to some members that are not declared in the base class (like the `objects` member). Python allows this, at least if the member is declared before being used. This is a *Django* class and we can't do too much here.

Known Issues

Bad URL when the webserver is not exposed

If the web server is run behind an Nginx proxy server the mail with the authentication token will be sent with the address of the *Django* (hidden) server, thus is not usable directly.

In such a case the user just have to adjust the hostname or ip to be the one of the exposed proxy.

Example of an email coming from a django server running in the docker-compose private network:

`http://web:8000/accounts/login?token=b92d59a9-9150-4b46-8a60-6cf0e5e481ca`

Just replace the "`web`" string with the ip/hostname of the exposed machine (e.g. localhost).

Obviously this "issue" can be fixed by let the *Django* server knows that he runs behind a proxy, so that he can put the address of the proxy in place of its address in the email.

The same problem is obtained in all the cases where the web server is not exposed directly to the client network, for example it is in a natted subnetwork.

Additional notes

Test client threads when debugging

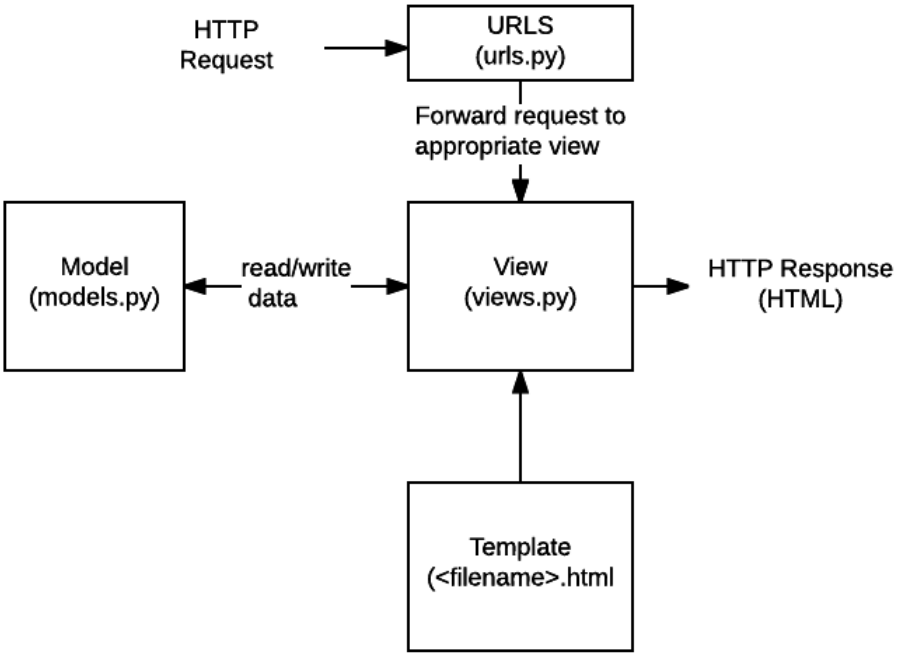
Assume you're debugging a functional tests and the *Django* test client requires to execute a view function where you've already placed a breakpoint, if you step through the call then will look like the debugger is frozen.

Paying more attention to the Eclipse threads lists we can see that the debugger is waiting for us in the view function, but in another thread. The thread must be changed manually from the list (at least in my Eclipse PyDev plugin).

Appendix 1: Django overview

In a traditional data-driven website, a web application waits for HTTP requests from the web browser (or other client). When a request is received the application works out what is needed based on the URL and possibly information in `POST` data or `GET` data. Depending on what is required it may then read or write information from a database or perform other tasks required to satisfy the request. The application will then return a response to the web browser, often dynamically creating an HTML page for the browser to display by inserting the retrieved data into placeholders in an HTML template.

Django web applications typically group the code that handles each of these steps into separate files:



URLs. Mapper is used to redirect HTTP requests to the appropriate view based on the request URL. The URL mapper can also match particular patterns of strings or digits that appear in an URL, and pass these to a view function as parameters arguments.

Views. Request handler functions, which receives HTTP requests and returns HTTP responses. Views access the data needed to satisfy requests via *models*, and delegate the formatting of the response to *templates*.

Models. Python objects that define the structure of an application's data, and provide mechanisms to create, remove, update and delete (CRUD) records in the database.

Templates. Text file defining the structure or layout of an HTML page, with placeholders used to represent actual content. A *view* can dynamically create an HTML page using an HTML template, populating it with data from a *model*.

Django refers to this organisation as the "*Model View Template"* (MVT) architecture. It has many similarities to the more familiar "*Model View Controller*" (MVC) architecture.

Mini-tutorial: Installation and Project Creation

Django can be easily installed via Pip.

```
$ pip install django
```

Once that *Django* has been installed, we create a new project with the "`django-admin startproject`" command.

```
$ django-admin startproject toddo
```

The command creates a minimal project files with the following structure:

```
toddo
├── manage.py
└── toddo
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

The **`manage.py`** python script is used to give "administrative" commands. It allows us to create a new applications, start the built-in test server, create the database, start the tests runner.

We proceed creating an application. Lists:

```
$ python manage.py startapp lists .
```

Follows the resulting project tree, more similar to the one of our *ToDDo* application (without the authentication subsystem).

```
toddo
```

```
├── lists
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── toddo
    ├── __init__.py
    ├── __pycache__
    │   ├── __init__.cpython-36.pyc
    │   └── settings.cpython-36.pyc
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

In the tree we can recognize some of the components that are part of the illustrated (MVT) model. The template directory should be created by hand, we've decided to create it under the lists application directory (i.e. `toddo/lists/templates`).

Appendix 2: Quickstart

Assuming you've already installed git, the first step is clone the toddo repository form *GitHub*.

```
$ git clone https://github.com/davxy/toddo.git
```

Docker compose

For a quickstart we'll run everything via docker and docker-compose. First install docker-compose, *Docker* will be installed as a dependency.

```
$ sudo apt-install docker-compose
```

Add your user to the docker group

```
$ sudo usermod -a -G docker <user>
```

Logout and login to let the running user be effectivelly viewed as part of the docker group. Optionally, you can use the -l su option:

```
$ su -l <user>
```

Move to the toddo project tools directory

```
$ cd toddo/tools
```

Start docker compose, that will eventually download the missing images before run the containers

```
docker-compose up
```

The docker-compose script is given below

tools/docker-compose.yml

```
version: '2'
services:
    nginx:
        image: nginx:latest
        volumes:
            - "./nginx-config:/etc/nginx/conf.d"
        ports:
            - "8000:80"
        depends_on:
```

```
            - web
    web:
        image: davxy/django
        command: toddo/django-start.sh
        volumes:
            - "..:/toddo"
        environment:
            - EMAIL_PASSWORD=changeme
```

In short, it starts two docker containers, nginx and web. The nginx container acts as a proxy to the internal django web server exposing the port 8000 to the world. The `django-start.sh` script creates the django database and starts the server to listen on port 8000.

```
Outside Network                Private Net
     <------>(8000)[nginx](80)<--------->(8000)[web]
```

Warning

If you want to test to let the emails be correctly sent to the users then you need to properly set the `EMAIL_PASSWORD` environment variable within the docker container. If you run the service from docker-compose tool then just change the `EMAIL_PASSWORD` setting within the `docker-compose.yml` file to the proper (secret) value… just ask me[3].

The quicker alternative is to directly alterate the `EMAIL_HOST_PASSWORD` in the "`toddo/settings.py`" file to be the correct password instead to read it from the environment.

Custom script: docker-run.sh

The script is not meant to substitute the docker-compose potential (e.g. private networks and dependency orchestration) but it offers an alternative *quick-and-dirty* way to run the application in five different ways within a single container

```
$ ./docker-run -t django|nginx|gunico|shell|test
```

- "`django`" : runs a stand alone django built-in server and expose the port 8000. The container uses the "`davxy/django`" image.

- "`nginx`" : runs a django server that listens on the docker localhost. The port 8000 is exposed outside by using an Nginx proxy server. The container uses the "`davxy/nginx`" image.

3 The alternative is to change the `EMAIL_HOST` and the `EMAIL_HOST_USER` the `toddo/settings.py` to an account you own. This could be the best option in the long run.

- “`gunico`” : runs the *Django* project by using a *Gunicorn* server. Application data is exchanged with an Nginx proxy via a Unix socket. The *Nginx* proxy server is exposed outside and listens on port 8000. The container uses the “`davxy/gunicorn`” image.

- “`test`” : runs all the tests of the *Django* project within the container. The container uses the “`davxy/django-test`” image, an extension of the “`selenium/standalone-chrome`”.

- “`shell`” : starts an interactive shell within the `davxy/django-test` container, the user is free to start the django server, run tests and a like.

The docker-compose method, implemented by the provided configuration file, is similar to the “`nginx`” option but uses two separate containers that communicates together via a private network created by docker compose tool. The docker-run script, starts the two services on the same host.

Run on local machine

Even if the simplest way to run the service is within a docker machine, some users may prefer to run the service directly within the host.

In such a case, first install all the required dependencies

```
$ sudo apt update && sudo apt install python3 python3-pip sqlite3
$ sudo ln -s /usr/bin/python3 /usr/bin/python
$ sudo ln -s /usr/bin/pip3 /usr/bin/pip
```

Next, move into the *ToDDo* project root folder, and build the database and required tables

```
$ python manage.py migrate
```

Now you should be able to run the *Django* server with the following commands

```
$ python manage.py runserver
```

Open a web-browser and point it to the following URL: http://localhost:8000.

To be able to run the functional tests you also need to install some additional system packages and python libraries:

```
$ apt install xvfb
$ pip install setuptools
$ pip install django==2.0 selenium pyvirtualdisplay
```

Eventually download and export to the curren environment the *selenium* drivers:

```
$ tools/getdrivers.py
$ export PATH=$PATH:$PWD/drivers/bin
```

All the project unit and functional tests can be run in one shot with the following command:

```
$ python manage.py test -v2
```

Actually the functional tests are created to run with both the *Firefox* and the *Chrome* web browsers. If you don’t have one of the mentioned browser within your machine, you can skip the tests for one application by defining the proper variable (refer to the “functional tests” paragraph of the “Test Driven Development” section for details)