**Low Level Design**

Blog Creator Web Application

|  |  |
| --- | --- |
| Written By | Akshay Sharma |
| Document Version | 0.3 |
| Last Revised Date | 17 – Dec -2022 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Author** | **Comments** |
| 0.1 | 12 – Dec -2022 | Akshay Sharma | Introduction & Architecture defined |
| 0.2 | 13 – Dec -2022 | Akshay Sharma | Architecture & Architecture Description appended and updated |
| 0.3 | 15 – Dec -2022 | Akshay Sharma | Unit Test Cases defined and appended |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

1. Introduction\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

* 1. What is Low-Level design document? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 1
  2. Scope\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1

1. Architecture\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2
2. Architecture Description\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3
   1. Django Admin\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3
   2. Create Project\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3
   3. URLs\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3
   4. Views\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3
   5. Models\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3
   6. Migrations \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3
   7. Register Admin \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3

3.8. Create Superuser \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_4

3.9. Ckeditor\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_4

3.10. Templates\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_4

3.11. Take Input\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_4

3.12. Save On Sqlite\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_4

1. Unit Test Cases \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5

1. Introduction

1.1. What is Low-Level design document?

The goal of LLD or a low-level design document (LLDD) is to give the internal logical design of the

actual program code for Food Recommendation System. LLD describes the class diagrams with the

methods and relations between classes and program specs. It describes the modules so that the

programmer can directly code the program from the document.

1.2. Scope

Low-level design (LLD) is a component-level design process that follows a step-by-

step refinement process. This process can be used for designing data structures, required software

architecture, source code and ultimately, performance algorithms. Overall, the data organization

may be defined during requirement analysis and then refined during data design work

2. Architecture

Create Project

Django-admin Start project

URLs

Migrations

Views

Models

Register Adimin

Ckeditor

Create Superuser

End

Templates

Save on SQLite

Take input

3. Architecture Description

3.1. Django Admin

check

compile messages

createcachetable

dbshell

diffsettings

dumpdata

flush

inspectdb

loaddata

makemessages

makemigrations

migrate

optimizemigration

runserver

sendtestemail

shell

showmigrations

sqlflush

sqlmigrate

sqlsequencereset

squashmigrations

startapp

startproject

test

testserver

3.2. Create Project

If this is your first-time using Django, you’ll have to take care of some initial setup. Namely, you’ll need to auto-generate some code that establishes a Django [project](https://docs.djangoproject.com/en/4.1/glossary/#term-project) – a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

From the command line, **cd** into a directory where you’d like to store your code, then run the following command:

**$** django-admin startproject mysite

3.3. URLs

To design URLs for an app, you create a Python module informally called a **URLconf** (URL configuration). This module is pure Python code and is a mapping between URL path expressions to Python functions (your views).

This mapping can be as short or as long as needed. It can reference other mappings. And, because it’s pure Python code, it can be constructed dynamically.

3.4. Views

A view function, or view for short, is a Python function that takes a web request and returns a web response. This response can be the HTML contents of a web page, or a redirect, or a 404 error, or an XML document, or an image . . . or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response. This code can live anywhere you want, as long as it’s on your Python path. There’s no other requirement–no “magic,” so to speak. For the sake of putting the code somewhere, the convention is to put views in a file called views.py, placed in your project or application directory.

3.5. Models

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you’re storing. Generally, each model maps to a single database table.

The basics:

Each model is a Python class that subclasses django.db.models.Model.

Each attribute of the model represents a database field.

With all of this, Django gives you an automatically-generated database-access API

3.6. Migrations

Migrations are Django’s way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They’re designed to be mostly automatic, but you’ll need to know when to make migrations, when to run them, and the common problems you might run into

3.7. Register Admin

One of the most powerful parts of Django is the automatic admin interface. It reads metadata from your models to provide a quick, model-centric interface where trusted users can manage content on your site. The admin’s recommended use is limited to an organization’s internal management tool. It’s not intended for building your entire front end around.

The admin has many hooks for customization, but beware of trying to use those hooks exclusively. If you need to provide a more process-centric interface that abstracts away the implementation details of database tables and fields, then it’s probably time to write your own views.

In this document we discuss how to activate, use, and customize Django’s admin interface.

3.8. Create Superuser

**$** python manage.py createsuperuser

Generating admin sites for your staff or clients to add, change and delete content is tedious work that doesn’t require much creativity. For that reason, Django entirely automates creation of admin interfaces for models.

Django was written in a newsroom environment, with a very clear separation between “content publishers” and the “public” site. Site managers use the system to add news stories, events, sports scores, etc., and that content is displayed on the public site. Django solves the problem of creating a unified interface for site administrators to edit content.

The admin isn’t intended to be used by site visitors. It’s for site managers.

3.9. Ckeditor

Provides a RichTextField, RichTextUploadingField, CKEditorWidget and CKEditorUploadingWidget utilizing CKEditor with image uploading and browsing support included.

This version also includes:

1. support to django-storages (works with S3)
2. updated ckeditor to version 4.18.0
3. included all ckeditor language and plugin files to make everyone happy!

3.10. Templates

A Django template is a text document or a Python string marked-up using the Django template language. Some constructs are recognized and interpreted by the template engine. The main ones are variables and tags.

A template is rendered with a context. Rendering replaces variables with their values, which are looked up in the context, and executes tags. Everything else is output as is.

The syntax of the Django template language involves four constructs.

**Variables**[**¶**](https://docs.djangoproject.com/en/4.1/topics/templates/#variables)

A variable outputs a value from the context, which is a dict-like object mapping keys to values.

3.11. Take Input

In HTML, a form is a collection of elements inside **<form>...</form>** that allow a visitor to do things like enter text, select options, manipulate objects or controls, and so on, and then send that information back to the server.

Some of these form interface elements - text input or checkboxes - are built into HTML itself. Others are much more complex; an interface that pops up a date picker or allows you to move a slider or manipulate controls will typically use JavaScript and CSS as well as HTML form **<input>** elements to achieve these effects.

As well as its **<input>** elements, a form must specify two things:

* *where*: the URL to which the data corresponding to the user’s input should be returned
* *how*: the HTTP method the data should be returned by

As an example, the login form for the Django admin contains several **<input>** elements: one of **type="text"** for the username, one of **type="password"** for the password, and one of **type="submit"** for the “Log in” button. It also contains some hidden text fields that the user doesn’t see, which Django uses to determine what to do next.

It also tells the browser that the form data should be sent to the URL specified in the **<form>**’s **action** attribute - **/admin/** - and that it should be sent using the HTTP mechanism specified by the **method** attribute - **post**.

When the **<input type="submit" value="Log in">** element is triggered, the data is returned to **/admin/**.

**GET and POST**[**¶**](https://docs.djangoproject.com/en/4.1/topics/forms/#get-and-post)

**GET** and **POST** are the only HTTP methods to use when dealing with forms.

Django’s login form is returned using the **POST** method, in which the browser bundles up the form data, encodes it for transmission, sends it to the server, and then receives back its response.

**GET**, by contrast, bundles the submitted data into a string, and uses this to compose a URL. The URL contains the address where the data must be sent, as well as the data keys and values. You can see this in action if you do a search in the Django documentation, which will produce a URL of the form **https://docs.djangoproject.com/search/?q=forms&release=1**.

**GET** and **POST** are typically used for different purposes.

Any request that could be used to change the state of the system - for example, a request that makes changes in the database - should use **POST**. **GET** should be used only for requests that do not affect the state of the system.

**GET** would also be unsuitable for a password form, because the password would appear in the URL, and thus, also in browser history and server logs, all in plain text. Neither would it be suitable for large quantities of data, or for binary data, such as an image. A web application that uses **GET** requests for admin forms is a security risk: it can be easy for an attacker to mimic a form’s request to gain access to sensitive parts of the system. **POST**, coupled with other protections like Django’s [CSRF protection](https://docs.djangoproject.com/en/4.1/ref/csrf/) offers more control over access.

On the other hand, **GET** is suitable for things like a web search form, because the URLs that represent a **GET** request can easily be bookmarked, shared, or resubmitted.

3.12. Save on Sqlite

**Summary**: in this tutorial, you will learn how to insert rows into a table in the SQLite database from a Python program using the sqlite3 module.

To insert rows into a table in SQLite database, you use the following steps:

1. First, [connect to the SQLite database](https://www.sqlitetutorial.net/sqlite-python/creating-database/) by creating a Connection object.
2. Second, create a Cursor object by calling the cursor method of the Connection object.
3. Third, execute an [INSERT statement](https://www.sqlitetutorial.net/sqlite-insert/). If you want to pass arguments to the INSERT statement, you use the question mark (?) as the placeholder for each argument.