**DTH OPERATION CONTROLLER**

**ABSTRACT**

This abstract presents a comprehensive DTH (Direct-to-Home) operation controller developed using Django, a Python web framework. The system integrates essential functionalities such as subscriber management, billing, email notification, technical support, and payment using QR code. Through its intuitive interface and robust backend, the controller streamlines DTH service operations, enhances customer experience, and ensures efficient management of subscriber services. Leveraging Django's features, including ORM (Object-Relational Mapping) and authentication, the system facilitates seamless subscriber onboarding, billing automation, real-time email notifications, and responsive technical support. Additionally, the integration of QR code payment functionality enables secure and convenient payment processing. Overall, the DTH operation controller represents a sophisticated solution for optimizing DTH service management, catering to the evolving needs of both operators and subscribers.

**INTRODUCTION**

In the rapidly evolving landscape of Direct-to-Home (DTH) television services, efficient management and seamless operation are paramount to ensure customer satisfaction and operational excellence. This introduction presents a pioneering DTH operation controller developed using Django, a versatile web framework powered by Python. The system is designed to integrate key functionalities including subscriber management, billing, email notification, technical support, and payment processing using QR codes, thereby offering a comprehensive solution to streamline DTH service operations.

The DTH operation controller serves as a centralized platform for managing every aspect of DTH service provision, from subscriber onboarding to technical issue resolution. Its user-friendly interface and robust backend infrastructure enable operators to efficiently handle subscriber requests, monitor service performance, and optimize resource allocation. Leveraging Django's advanced features, such as ORM (Object-Relational Mapping) and authentication, the system ensures secure, scalable, and responsive service delivery.

Key features of the DTH operation controller include:

1. Subscriber Management: Facilitating seamless onboarding, plan management, and database maintenance for subscribers.
2. Email Notification: Providing real-time email notifications for service activations, billing reminders, and technical support updates to enhance communication with subscribers.
3. Technical Support: Offering a robust resolving technical issues promptly, ensuring uninterrupted service delivery.
4. Payment using QR Code: Integrating QR code functionality for secure and convenient payment processing, enhancing the payment experience for subscribers.

By combining these essential functionalities within a single platform, the DTH operation controller optimizes operational efficiency, improves customer experience, and fosters sustainable growth for DTH service providers. Its modular architecture and flexible design allow for seamless integration with existing DTH infrastructure and third-party systems, ensuring compatibility and interoperability.

In summary, the DTH operation controller developed using Django represents a groundbreaking solution for managing DTH service operations effectively and efficiently. By harnessing the power of technology and innovation, it empowers operators to deliver superior service quality, drive customer satisfaction, and thrive in the competitive DTH market landscape.

**Existing System with Drawbacks**

Fragmented Systems: Many existing DTH operation management systems rely on fragmented solutions for subscriber management, billing, email notification, technical support, and payment processing. This fragmentation can lead to disjointed workflows, data duplication, and increased operational complexity.

Manual Processes: Some systems still heavily rely on manual processes for subscriber management, billing, and technical support. Manual data entry and processing increase the likelihood of errors, delays, and inefficiencies, impacting both service quality and customer satisfaction.

Limited Automation: Existing systems may lack robust automation capabilities, particularly in billing, email notification, and technical support processes. This limitation results in manual intervention for routine tasks, consuming valuable time and resources that could be allocated elsewhere.

Inadequate Communication: Communication with subscribers may be inadequate in existing systems, with limited options for real-time email notifications and updates. This lack of communication can lead to customer frustration, confusion, and dissatisfaction with the service.

Security Concerns: Security concerns, particularly related to payment processing and data privacy, may pose significant challenges in existing systems. Without robust security measures in place, sensitive subscriber information and payment data are vulnerable to breaches and unauthorized access.

Limited Payment Options: Existing systems may offer limited payment options for subscribers, with traditional methods such as credit/debit cards dominating. The absence of modern payment solutions, such as QR code payments, restricts convenience and accessibility for subscribers.

Scalability Issues: Scalability can be a concern for existing systems, particularly as subscriber bases grow and service demands increase. Inflexible architectures and outdated technologies may struggle to accommodate the expanding needs of DTH operators, leading to performance issues and service disruptions.

Poor Technical Support: Technical support mechanisms in existing systems may be insufficient, with limited ticketing systems and response times. Inadequate technical support can result in prolonged service disruptions, unresolved issues, and dissatisfied customers.

**Proposed System for Advantages**

Subscriber Management: The system offers a centralized platform for managing subscriber accounts, including registration, plan selection, and account updates. Operators can easily onboard new subscribers, modify subscription plans, and maintain accurate subscriber databases.

Email Notification: Real-time email notifications keep subscribers informed about service activations, billing updates, and technical support responses. Customizable email templates allow operators to personalize communication and enhance subscriber engagement.

Technical Support: A robust ticketing system enables subscribers to report technical issues, track ticket status, and receive timely assistance from support staff. Operators can assign tickets, prioritize responses, and maintain a comprehensive history of support interactions.

Payment using QR Code: Integration of QR code payment functionality provides subscribers with a convenient and secure payment option. Subscribers can scan QR codes displayed on payment portals to initiate transactions, streamlining the payment process and enhancing user experience.

Advantages of the Proposed System:

Streamlined Operations: The integrated nature of the system streamlines DTH service operations, reducing manual intervention and improving operational efficiency.

Enhanced Customer Experience: Automated processes, real-time notifications, and convenient payment options enhance the overall customer experience, leading to higher satisfaction and loyalty.

Improved Communication: Personalized email notifications and responsive technical support mechanisms improve communication with subscribers, fostering stronger relationships and trust.

Secure Payment Processing: Integration of QR code payment ensures secure and seamless payment processing, reducing the risk of fraud and enhancing payment security.

Scalability and Flexibility: Built on Django, the system offers scalability and flexibility to accommodate growing subscriber bases and evolving service needs, ensuring long-term viability and adaptability.

**SYSTEM SPECIFICATION**

HARDWARE CONFIGURATION

Processor: Intel Core i3

RAM Capacity: 8 GB

Hard Disk: 1 TB

Mouse: Logical Optical Mouse

Keyboard: Logitech 107 Keys

Monitor: 14 inch

Mother Board: Intel

Speed: 3.3GHZ

SOFTWARE SPECIFICATION

Operating System: Windows 10

Back End: PYTHON

Workbench: Pycharm, VSCode

Front End: HTML, CSS, JS, Bootstrap

Framework: DJANGO

**ABOUT SOFTWARE**

**PYTHON**

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Python Features

Python has few keywords, simple structure, and a clearly defined syntax. Python code is more clearly defined and visible to the eyes. Python's source code is fairly easy-to-maintaining. Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh. Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.

Portable Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

Extendable

It allows to add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.

Databases

Python provides interfaces to all major commercial databases.

GUI Programming

Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

Scalable

Python provides a better structure and support for large programs than shell scripting.

Object-Oriented Approach

One of the key aspects of Python is its object-oriented approach. This basically means that Python recognizes the concept of class and object encapsulation thus allowing programs to be efficient in the long run.

Highly Dynamic

Python is one of the most dynamic languages available in the industry today. There isno need to specify the type of the variable during coding, thus saving time and increasing efficiency.

Extensive Array of Libraries

Python comes inbuilt with many libraries that can be imported at any instance and be used in a specific program.

Open Source and Free

Python is an open-source programming language which means that anyone can create and contribute to its development. Python is free to download and use in any operating system, like Windows, Mac or Lin.

MODULES

A module allows you to logically organize your Python code. Grouping related code into a module makes

the code easier to understand and use. A module is a Python object with arbitrarily named attributes that

you can bind and reference. Simply, a module is a file consisting of Python code. A module can define

functions, classes and variables. A module can also include runnable code.

Example:

The Python code for a module named aname normally resides in a file named aname.py. Here's an

example of a simple module, support.py

def print\_func( par ):

print "Hello : ", par

return

The importStatement

The import has the following syntax:

import module1[, module2[,... moduleN]

When the interpreter encounters an import statement, it imports the module if the module is present in the

search path. A search path is a list of directories that the interpreter searches before importing a module.

For example, to import the module support.py, you need to put the following command at the top of the

script −

A module is loaded only once, regardless of the number of times it is imported. This prevents the module

execution from happening over and over again if multiple imports occur.

Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment

that consists of modules and sub packages and sub-sub packages.

Consider a file Pots.py available in Phone directory. This file has following line of source code −

def Pots():

print "I'm Pots Phone"

Similar way, we have another two files having different functions with the same name as above −

• Phone/Isdn.py file having function Isdn()

• Phone/G3.py file having function G3()

Now, create one more file \_init\_.py in Phone directory −

• Phone/\_init\_.py

To make all of your functions available when you've imported Phone,to put explicit import statements in

\_init\_.py as follows −

from Pots import Pots

from Isdn import Isdn

from G3 import G3

After you add these lines to \_init\_.py, you have all of these classes available when you import the

Phone package.

# Now import your Phone Package.

import Phone

Phone.Pots()

Phone.Isdn()

Phone.G3()

RESULT:

I'm Pots Phone

I'm 3G Phone

I'm ISDN Phone

In the above example, we have taken example of a single functions in each file, but you can keep multiple

functions in your files. You can also define different Python classes in those files and then you can create

your packages out of those classes.

PYTHON FILES I/O

This chapter covers all the basic I/O functions available in Python.

PRINTING TO THE SCREEN

The simplest way to produce output is using the print statement where you can pass zero or more

expressions separated by commas. This function converts the expressions you pass into a string and

writes the result to standard output as follows −

print "Python is really a great language,", "isn't it?"

Result:

Python is really a great language, isn't it?

READING KEYBOARD INPUT

Python provides two built-in functions to read a line of text from standard input, which by default comes

from the keyboard. These functions are −

• raw\_input

• input

Theraw\_inputFunction

The raw\_input([prompt]) function reads one line from standard input and returns it as a string (removing

the trailing newline).

str = raw\_input("Enter your input: ");

print "Received input is : ", str

This prompts you to enter any string and it would display same string on the screen. When I typed "Hello

Python!", its output is like this −

Enter your input: Hello Python

Received input is : Hello Python

22 | P a g e

The input Function

The input([prompt]) function is equivalent to raw\_input, except that it assumes the input is a valid Python

expression and returns the evaluated result to you.

str = input("Enter your input: ");

print "Received input is : ", str

This would produce the following result against the entered input −

Enter your input: [x\*5 for x in range(2,10,2)]

Recieved input is : [10, 20, 30, 40]

Class and Object

Python has been an object-oriented language since it existed. Because of this, creating and using classes

and objects are downright easy. This chapter helps you become an expert in using Python's objectoriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to

consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic

concepts.

However, here is small introduction of Object-Oriented Programming (OOP) to bring you at speed −

Overview of OOP Terminology

• Class: A user-defined prototype for an object that defines a set of attributes that characterize

any object of the class. The attributes are data members (class variables and instance

variables) and methods, accessed via dot notation.

• Class variable: A variable that is shared by all instances of a class. Class variables are defined

within a class but outside any of the class's methods. Class variables are not used as frequently

as instance variables are.

• Data member: A class variable or instance variable that holds data associated with a class

and its objects.

• Function overloading: The assignment of more than one behavior to a particular function.

The operation performed varies by the types of objects or argument

• Instance variable: A variable that is defined inside a method and belongs only to the current

instance of a class.

• Inheritance: The transfer of the characteristics of a class to other classes that are derived from

it.

• Instance: An individual object of a certain class. An object obj that belongs to a class Circle,

for example, is an instance of the class Circle.

• Instantiation: The creation of an instance of a class.

• Method : A special kind of function that is defined in a class definition.

Object: A unique instance of a data structure that's defined by its class. An object comprises

both data members (class variables and instance variables) and methods.

• Operator overloading: The assignment of more than one function to a particular operator.

Creating Classes

The class statement creates a new class definition. The name of the class immediately follows the

keyword class followed by a colon as follows −

class ClassName:

'Optional class documentation string'

class\_suite

• The class has a documentation string, which can be accessed via ClassName.\_doc\_.

• The class\_suite consists of all the component statements defining class members, data

attributes and functions.–––

Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing

the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were

defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from

is given after the class name −

class SubClassName (ParentClass1[, ParentClass2, ...]):

'Optional class documentation string'

class\_suite

Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is

because you may want special or different functionality in your subclass.

Example

class Parent: # define parent class

def myMethod(self):

print 'Calling parent method'

class Child(Parent): # define child class

def myMethod(self):

print 'Calling child method'

c = Child() # instance of child

c.myMethod() # child calls overridden method

When the above code is executed, it produces the following result −

Calling child method

Base Overloading Methods

Following table lists some generic functionality that you can override in your own classes −

SN Method, Description & Sample Call

1 \_init\_ ( self [,args...] )

Constructor (with any optional arguments)

Sample Call : obj = className(args)

2 \_del\_( self )

Destructor, deletes an object

Sample Call : del obj

3 \_repr\_( self )

Evaluatable string representation

Sample Call : repr(obj)

4 \_str\_( self )

Printable string representation

Sample Call : str(obj)

5 \_cmp\_ ( self, x )

Object comparison

Sample Call : cmp(obj, x)

Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you

use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the \_add\_ method in your class to perform vector addition and then the

plus operator would behave as per expectation −

class Vector:

def \_init\_(self, a, b):

self.a = a

self.b = b

def \_str\_(self):

return 'Vector (%d, %d)' % (self.a, self.b)

def \_add\_(self,other):

return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)

v2 = Vector(5,-2)

print v1 + v2

When the above code is executed, it produces the following result –

Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes

with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

lass JustCounter:

\_\_secretCount = 0

def count(self):

self.\_\_secretCount += 1

print self.\_\_secretCount

counter = JustCounter()

counter.count()

counter.count()

print counter.\_\_secretCount

1

2

Traceback (most recent call last):

File "test.py", line 12, in <module>

print counter.\_\_secretCount

AttributeError: JustCounter instance has no attribute '\_\_secretCount'

Python protects those members by internally changing the name to include the class name. You can

access such attributes as object.className\_attrName. If you would replace your last line as following,

then it works for you −

.........................

**Django**

**Overview:**

Design your model[**¶**](https://docs.djangoproject.com/en/4.2/intro/overview/#design-your-model)

Although you can use Django without a database, it comes with an [object-relational mapper](https://en.wikipedia.org/wiki/Object-relational_mapping) in which you describe your database layout in Python code.

The [data-model syntax](https://docs.djangoproject.com/en/4.2/topics/db/models/) offers many rich ways of representing your models – so far, it’s been solving many years’ worth of database-schema problems. Here’s a quick example:

mysite/news/models.py[**¶**](https://docs.djangoproject.com/en/4.2/intro/overview/#id1)

**from** **django.db** **import** models

**class** **Reporter**(models.Model):

full\_name = models.CharField(max\_length=70)

**def** \_\_str\_\_(self):

**return** self.full\_name

**class** **Article**(models.Model):

pub\_date = models.DateField()

headline = models.CharField(max\_length=200)

content = models.TextField()

reporter = models.ForeignKey(Reporter, on\_delete=models.CASCADE)

**def** \_\_str\_\_(self):

**return** self.headline

Install it[**¶**](https://docs.djangoproject.com/en/4.2/intro/overview/#install-it)

Next, run the Django command-line utilities to create the database tables automatically:

/ 

**$** python manage.py makemigrations

**$** python manage.py migrate

The **[makemigrations](https://docs.djangoproject.com/en/4.2/ref/django-admin/" \l "django-admin-makemigrations)** command looks at all your available models and creates migrations for whichever tables don’t already exist. [**migrate**](https://docs.djangoproject.com/en/4.2/ref/django-admin/#django-admin-migrate) runs the migrations and creates tables in your database, as well as optionally providing [much richer schema control](https://docs.djangoproject.com/en/4.2/topics/migrations/).

Enjoy the free API[**¶**](https://docs.djangoproject.com/en/4.2/intro/overview/#enjoy-the-free-api)

With that, you’ve got a free, and rich, [Python API](https://docs.djangoproject.com/en/4.2/topics/db/queries/) to access your data. The API is created on the fly, no code generation necessary:

# Import the models we created from our "news" app

**>>> from** **news.models** **import** Article, Reporter

# No reporters are in the system yet.

**>>>** Reporter.objects.all()

<QuerySet []>

# Create a new Reporter.

**>>>** r = Reporter(full\_name="John Smith")

# Save the object into the database. You have to call save() explicitly.

**>>>** r.save()

# Now it has an ID.

**>>>** r.id

1

# Now the new reporter is in the database.

**>>>** Reporter.objects.all()

<QuerySet [<Reporter: John Smith>]>

# Fields are represented as attributes on the Python object.

**>>>** r.full\_name

'John Smith'

# Django provides a rich database lookup API.

**>>>** Reporter.objects.get(id=1)

<Reporter: John Smith>

**>>>** Reporter.objects.get(full\_name\_\_startswith="John")

<Reporter: John Smith>

**>>>** Reporter.objects.get(full\_name\_\_contains="mith")

<Reporter: John Smith>

**>>>** Reporter.objects.get(id=2)

Traceback (most recent call last):

...

DoesNotExist: Reporter matching query does not exist.

# Create an article.

**>>> from** **datetime** **import** date

**>>>** a = Article(

**...**  pub\_date=date.today(), headline="Django is cool", content="Yeah.", reporter=r

**...** )

**>>>** a.save()

# Now the article is in the database.

**>>>** Article.objects.all()

<QuerySet [<Article: Django is cool>]>

# Article objects get API access to related Reporter objects.

**>>>** r = a.reporter

**>>>** r.full\_name

'John Smith'

# And vice versa: Reporter objects get API access to Article objects.

**>>>** r.article\_set.all()

<QuerySet [<Article: Django is cool>]>

# The API follows relationships as far as you need, performing efficient

# JOINs for you behind the scenes.

# This finds all articles by a reporter whose name starts with "John".

**>>>** Article.objects.filter(reporter\_\_full\_name\_\_startswith="John")

<QuerySet [<Article: Django is cool>]>

# Change an object by altering its attributes and calling save().

**>>>** r.full\_name = "Billy Goat"

**>>>** r.save()

# Delete an object with delete().

**>>>** r.delete()

A dynamic admin interface: it’s not just scaffolding – it’s the whole house

Once your models are defined, Django can automatically create a professional, production ready [administrative interface](https://docs.djangoproject.com/en/4.2/ref/contrib/admin/) – a website that lets authenticated users add, change and delete objects. The only step required is to register your model in the admin site:

mysite/news/models.py

**from** **django.db** **import** models

**class** **Article**(models.Model):

pub\_date = models.DateField()

headline = models.CharField(max\_length=200)

content = models.TextField()

reporter = models.ForeignKey(Reporter, on\_delete=models.CASCADE)

mysite/news/admin.py

**from** **django.contrib** **import** admin

**from** **.** **import** models

admin.site.register(models.Article)

The philosophy here is that your site is edited by a staff, or a client, or maybe just you – and you don’t want to have to deal with creating backend interfaces only to manage content.

One typical workflow in creating Django apps is to create models and get the admin sites up and running as fast as possible, so your staff (or clients) can start populating data. Then, develop the way data is presented to the public.

Design your URLs

A clean, elegant URL scheme is an important detail in a high-quality web application. Django encourages beautiful URL design and doesn’t put any cruft in URLs, like **.php** or **.asp**.

To design URLs for an app, you create a Python module called a [URLconf](https://docs.djangoproject.com/en/4.2/topics/http/urls/). A table of contents for your app, it contains a mapping between URL patterns and Python callback functions. URLconfs also serve to decouple URLs from Python code.

Here’s what a URLconf might look like for the **Reporter**/**Article** example above:

mysite/news/urls.py

**from** **django.urls** **import** path

**from** **.** **import** views

urlpatterns = [

path("articles/<int:year>/", views.year\_archive),

path("articles/<int:year>/<int:month>/", views.month\_archive),

path("articles/<int:year>/<int:month>/<int:pk>/", views.article\_detail),

]

The code above maps URL paths to Python callback functions (“views”). The path strings use parameter tags to “capture” values from the URLs. When a user requests a page, Django runs through each path, in order, and stops at the first one that matches the requested URL. (If none of them matches, Django calls a special-case 404 view.) This is blazingly fast, because the paths are compiled into regular expressions at load time.

Once one of the URL patterns matches, Django calls the given view, which is a Python function. Each view gets passed a request object – which contains request metadata – and the values captured in the pattern.

For example, if a user requested the URL “/articles/2005/05/39323/”, Django would call the function **news.views.article\_detail(request, year=2005, month=5, pk=39323)**.

Write your views

Each view is responsible for doing one of two things: Returning an **[HttpResponse](https://docs.djangoproject.com/en/4.2/ref/request-response/" \l "django.http.HttpResponse" \o "django.http.HttpResponse)** object containing the content for the requested page, or raising an exception such as [**Http404**](https://docs.djangoproject.com/en/4.2/topics/http/views/#django.http.Http404). The rest is up to you.

Generally, a view retrieves data according to the parameters, loads a template and renders the template with the retrieved data. Here’s an example view for **year\_archive** from above:

mysite/news/views.py

**from** **django.shortcuts** **import** render

**from** **.models** **import** Article

**def** year\_archive(request, year):

a\_list = Article.objects.filter(pub\_date\_\_year=year)

context = {"year": year, "article\_list": a\_list}

**return** render(request, "news/year\_archive.html", context)

This example uses Django’s [template system](https://docs.djangoproject.com/en/4.2/topics/templates/), which has several powerful features but strives to stay simple enough for non-programmers to use.

Design your templates

The code above loads the **news/year\_archive.html** template.

Django has a template search path, which allows you to minimize redundancy among templates. In your Django settings, you specify a list of directories to check for templates with [**DIRS**](https://docs.djangoproject.com/en/4.2/ref/settings/#std-setting-TEMPLATES-DIRS). If a template doesn’t exist in the first directory, it checks the second, and so on.

Let’s say the **news/year\_archive.html** template was found. Here’s what that might look like:

mysite/news/templates/news/year\_archive.html

{% **extends** "base.html" %}

{% **block** title %}Articles for {{ year }}{% **endblock** %}

{% **block** content %}

<**h1**>Articles for {{ year }}</**h1**>

{% **for** article **in** article\_list %}

<**p**>{{ article.headline }}</**p**>

<**p**>By {{ article.reporter.full\_name }}</**p**>

<**p**>Published {{ article.pub\_date|date:"F j, Y" }}</**p**>

{% **endfor** %}

{% **endblock** %}

Variables are surrounded by double-curly braces. **{{ article.headline }}** means “Output the value of the article’s headline attribute.” But dots aren’t used only for attribute lookup. They also can do dictionary-key lookup, index lookup and function calls.

Note **{{ article.pub\_date|date:"F j, Y" }}** uses a Unix-style “pipe” (the “|” character). This is called a template filter, and it’s a way to filter the value of a variable. In this case, the date filter formats a Python datetime object in the given format (as found in PHP’s date function).

You can chain together as many filters as you’d like. You can write [custom template filters](https://docs.djangoproject.com/en/4.2/howto/custom-template-tags/#howto-writing-custom-template-filters). You can write [custom template tags](https://docs.djangoproject.com/en/4.2/howto/custom-template-tags/), which run custom Python code behind the scenes.

Finally, Django uses the concept of “template inheritance”. That’s what the **{% extends "base.html" %}** does. It means “First load the template called ‘base’, which has defined a bunch of blocks, and fill the blocks with the following blocks.” In short, that lets you dramatically cut down on redundancy in templates: each template has to define only what’s unique to that template.

Here’s what the “base.html” template, including the use of [static files](https://docs.djangoproject.com/en/4.2/howto/static-files/), might look like:

mysite/templates/base.html

{% **load** static %}

<**html**>

<**head**>

<**title**>{% **block** title %}{% **endblock** %}</**title**>

</**head**>

<**body**>

<**img** src="{% **static** 'images/sitelogo.png' %}" alt="Logo">

{% **block** content %}{% **endblock** %}

</**body**>

</**html**>

Simplistically, it defines the look-and-feel of the site (with the site’s logo), and provides “holes” for child templates to fill. This means that a site redesign can be done by changing a single file – the base template.

It also lets you create multiple versions of a site, with different base templates, while reusing child templates. Django’s creators have used this technique to create strikingly different mobile versions of sites by only creating a new base template.

Note that you don’t have to use Django’s template system if you prefer another system. While Django’s template system is particularly well-integrated with Django’s model layer, nothing forces you to use it. For that matter, you don’t have to use Django’s database API, either. You can use another database abstraction layer, you can read XML files, you can read files off disk, or anything you want. Each piece of Django – models, views, templates – is decoupled from the next.

This is just the surface

This has been only a quick overview of Django’s functionality. Some more useful features:

* A [caching framework](https://docs.djangoproject.com/en/4.2/topics/cache/) that integrates with memcached or other backends.
* A [syndication framework](https://docs.djangoproject.com/en/4.2/ref/contrib/syndication/) that lets you create RSS and Atom feeds by writing a small Python class.
* More attractive automatically-generated admin features – this overview barely scratched the surface.

**SYSTEM STUDY**

System study contains existing and proposed system details. Existing system is useful to develop proposed system. To elicit the requirements of the system and to identify the elements, Inputs, Outputs, subsystems and the procedures, the existing system had to be examined and analyzed in detail.

This increases the total productivity. The use of paper files is avoided and all the data are efficiently manipulated by the system. It also reduces the space needed to store the larger paper files and records.

**SYSTEM DESIGN**

The degree of interest in each concept has varied over the year, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Fundamental design concepts provide the necessary framework for “getting it right”.

During the design process the software requirements model is transformed into design models that describe the details of the data structures, system architecture, interface, and components. Each design product is reviewed for quality before moving to the next phase of software development.

**INPUT DESIGN**

The design of input focus on controlling the amount of dataset as input required, avoiding delay and keeping the process simple. The input is designed in such a way to provide security. Input design will consider the following steps:

 The dataset should be given as input.

 The dataset should be arranged.

 Methods for preparing input validations.

**OUTPUT DESIGN**

A quality output is one, which meets the requirement of the user and presents the information clearly. In output design, it is determined how the information is to be displayed for immediate need.

Designing computer output should proceed in an organized, well thought out manner the right output must be developed while ensuring that each output element is designed so that the user will find the system can be used easily and effectively.

**DATABASE DESIGN**

This phase contains the attributes of the dataset which are maintained in the database table. The dataset collection can be of two types namely train dataset and test dataset.

**SYSTEM TESTING**

System testing was done by giving different training and testing datasets. This test was done to evaluate whether the system was predicting accurate result or not. During the phase of the development of the system our system was tested time and again.

The series of testing conducted are as follows:

**UNIT TESTING**

In unit testing, we designed the whole system in modularized pattern and each module was tested. Till we get the accurate output from the individual module we worked on the same module.

**INTEGRATION TESTING**

After constructing individual modules all the modules were merged and a complete system was made. Then the system was tested whether the prediction given by training dataset to testing set was correct or not. We tried to meet the accuracy as higher as much as we can get. After spending a couple of days in integration testing the average accuracy of our system was 91%.

**ALPHA TESTING**

Alpha testing is the first stage of software engineering which is considered as a simulated or actual operational testing done by the individual member of the project. Alpha testing is conducted by the project developers, in context of our project.

**BETA TESTING**

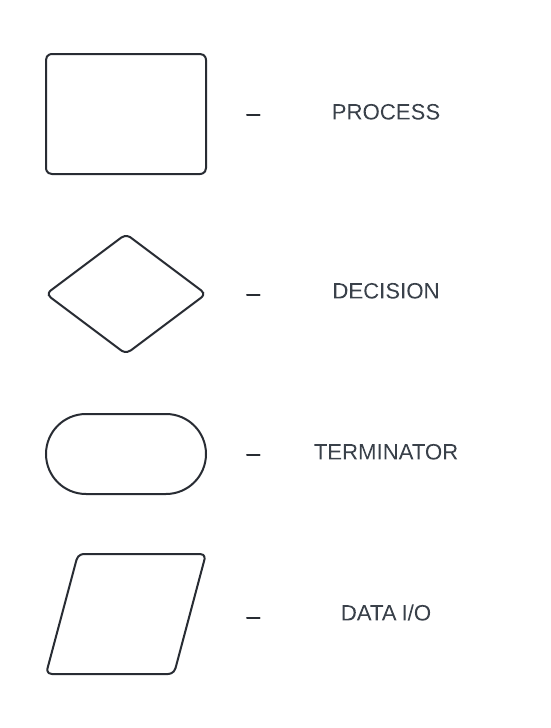
Beta testing comes continuously after alpha testing which is considered as a form of external user acceptance testing. The beta version of the program is developed to and provided to limited audience. This is the final test process in the case of this project. In this system the beta-testing is done by our colleagues and the project supervisor.

**DATAFLOW DIAGRAM**

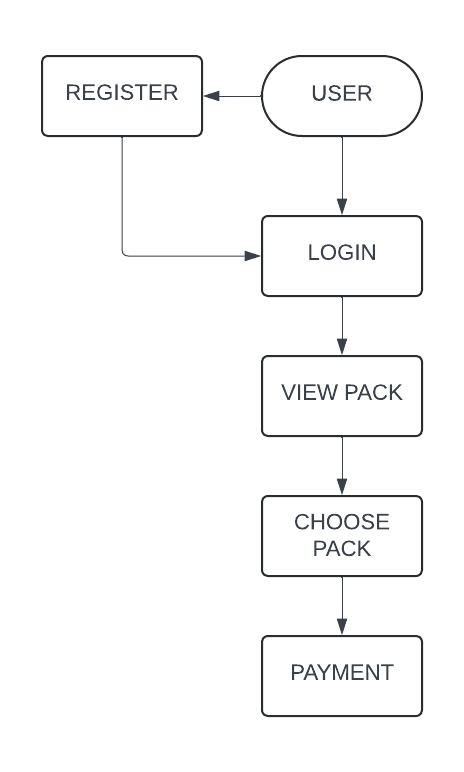
Data flow diagrams are used to graphically represent the flow of data in a business information system. DFD describes the processes that are involved in a system to transfer data from the input to the file storage and reports generation. Data flow diagrams can be divided into logical and physical. The logical data flow diagram describes flow of data through a system to perform certain functionality of a business. The physical data flow diagram describes the implementation of the logical data flow.

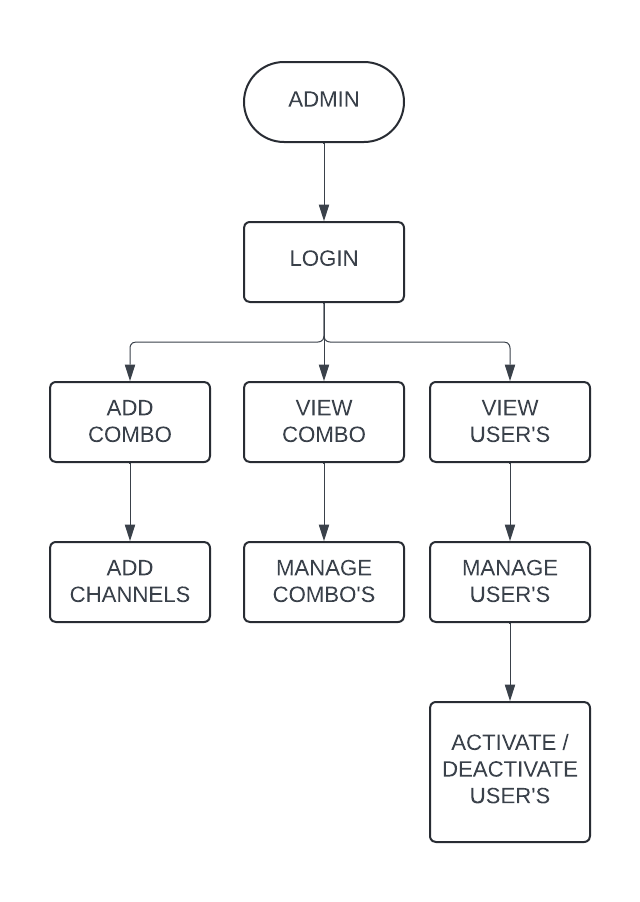
DFD graphically representing the functions, or processes, which capture, manipulate, store, and distribute data between a system and its environment and between components of a system. The visual representation makes it a good communication tool between User and System designer. The objective of a DFD is to show the scope and boundaries of a system. The DFD is also called as a data flow graph or bubble chart. It can be manual, automated, or a combination of both. It shows how data enters and leaves the system, what changes the information, and where data is stored.

Design Notation

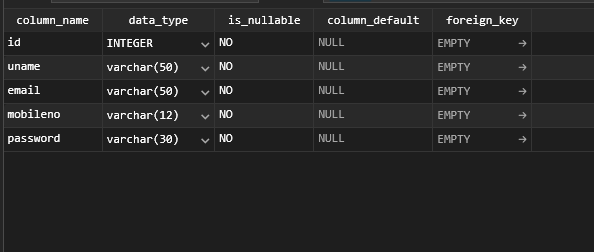
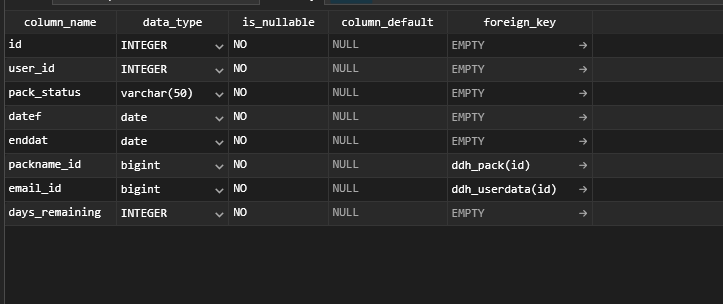
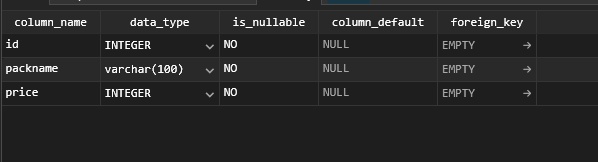
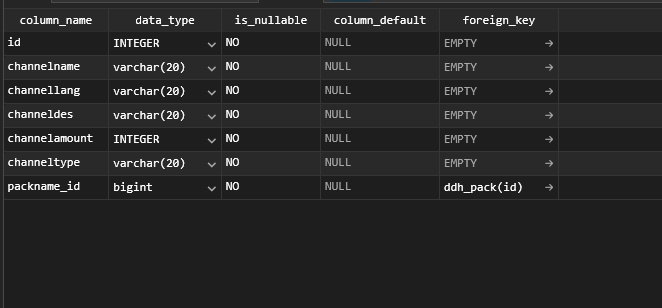
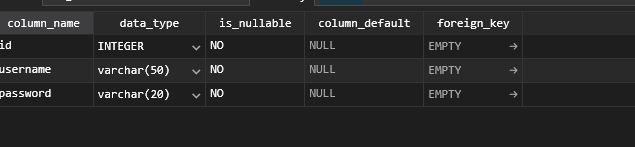


**DATAFLOW DIAGRAM**

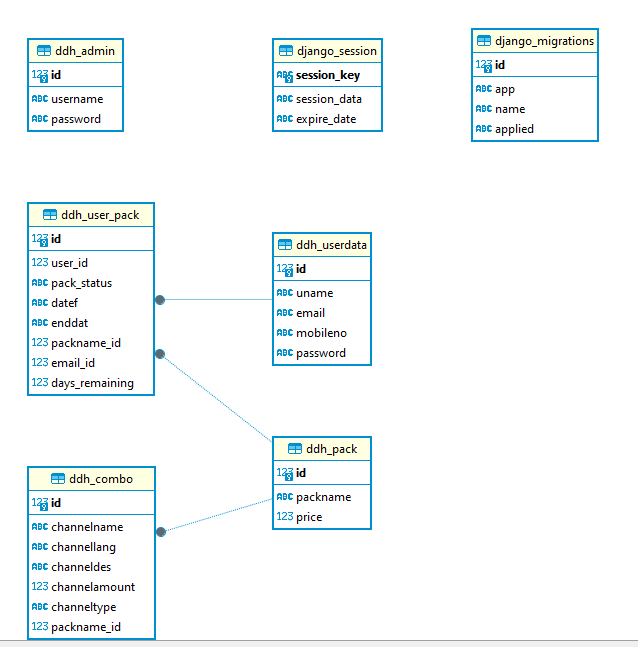




**TABLE DESIGN:**

****

**ER DIARAM:**

****

**MODULE DESCRIPTION**

Subscriber Management Module:

This module handles all aspects related to subscriber management, including registration, account creation, and plan selection.

Features include subscriber profile management, subscription plan modification.

Operators can access subscriber information, track subscription history, and manage subscription renewals.

Email Notification Module:

The email notification module sends real-time notifications to subscribers regarding service activations, billing updates, and technical support responses.

Operators can configure email notifications, track email delivery, and analyze subscriber engagement.

Technical Support Module:

The technical support module provides a robust customer care system for managing subscriber inquiries, technical issues, and support requests.

Payment using QR Code Module:

The payment using QR code module enables subscribers to make payments securely and conveniently using QR codes.

Features include QR code generation, payment processing.

Subscribers can scan QR codes displayed on payment portals to initiate transactions, simplifying the payment process.

**FUTURE ENHANCEMENT**

Enhanced Subscriber Management:

Implement advanced subscriber segmentation and targeting capabilities based on demographics, viewing preferences, and behavioral data.

Integrate machine learning algorithms to predict subscriber churn and proactively engage at-risk subscribers with personalized retention offers.

Intelligent Email Notification System:

Incorporate natural language processing (NLP) techniques to analyze subscriber feedback from email interactions and automate sentiment analysis.

Implement advanced email scheduling algorithms to optimize send times based on subscriber engagement patterns and time zones.

Augmented Technical Support:

Introduce AI-powered chatbots to handle basic technical support queries and assist subscribers in troubleshooting common issues autonomously.

Implement a knowledge base system with multimedia tutorials and self-help guides to empower subscribers to resolve issues independently.

Payment Security Enhancements:

Integrate biometric authentication options, such as fingerprint or facial recognition, for secure and convenient QR code payment authorization.

Implement blockchain technology for payment processing to enhance transparency, traceability, and security in transaction records.

Multi-Channel Customer Engagement:

Expand communication channels beyond email to include SMS, push notifications, and social media messaging for more personalized and omnichannel customer engagement.

Develop a mobile application for subscribers to access account information, receive real-time updates, and interact with support services on-the-go.

Predictive Analytics for Billing:

Utilize predictive analytics models to forecast future billing patterns, identify potential billing anomalies or discrepancies, and proactively address billing issues.

Implement dynamic pricing strategies based on subscriber behavior, consumption patterns, and market demand to optimize revenue generation.

Integration with IoT Devices:

Integrate with Internet of Things (IoT) devices, such as smart TVs and set-top boxes, to gather real-time usage data and personalize content recommendations for subscribers.

Enable voice-activated commands and control functionalities through IoT devices for seamless and intuitive user experiences.

Gamification and Rewards Program:

Introduce gamification elements into the platform, such as achievement badges, loyalty points, and leaderboard competitions, to incentivize subscriber engagement and reward loyalty.

Launch a rewards program that offers discounts, exclusive content access, and merchandise giveaways to subscribers based on their participation and engagement levels.

**CONCLUSION**

In conclusion, the DTH operation controller developed using Django represents a significant advancement in the management of Direct-to-Home (DTH) television services. By integrating subscriber management, email notification, technical support, and payment processing using QR code, the system offers a comprehensive solution to streamline DTH service operations and enhance customer experience.

Through robust subscriber management capabilities, operators can efficiently onboard new subscribers, manage subscription plans, and maintain accurate subscriber databases. Real-time email notifications keep subscribers informed about service activations, billing updates, and technical support responses, fostering better communication and engagement.

The technical support module provides a robust ticketing system for managing subscriber inquiries and technical issues, ensuring prompt resolution and high levels of customer satisfaction. Integration of QR code payment functionality offers secure and convenient payment processing options, enhancing the payment experience for subscribers.

In conclusion, the DTH operation controller developed using Django empowers operators to deliver superior service quality, drive customer satisfaction, and thrive in the competitive DTH market landscape. By leveraging advanced technologies and innovative features, the system sets a new standard for efficiency, effectiveness, and customer-centricity in DTH service operations.

**BIBLIOGRAPHY**

Web Sources:

* "DTH Subscriber Management System." Tata Sky, www.tatasky.com/subscriber-management-system.
* "Direct-to-Home (DTH) Subscriber Management System." Intelsat, www.intelsat.com/resources/dth-subscriber-management-system/.
* "QR Code Payment Systems: A Comprehensive Guide." PaymentSource, www.paymentsource.com/guide/qr-code-payment-systems.

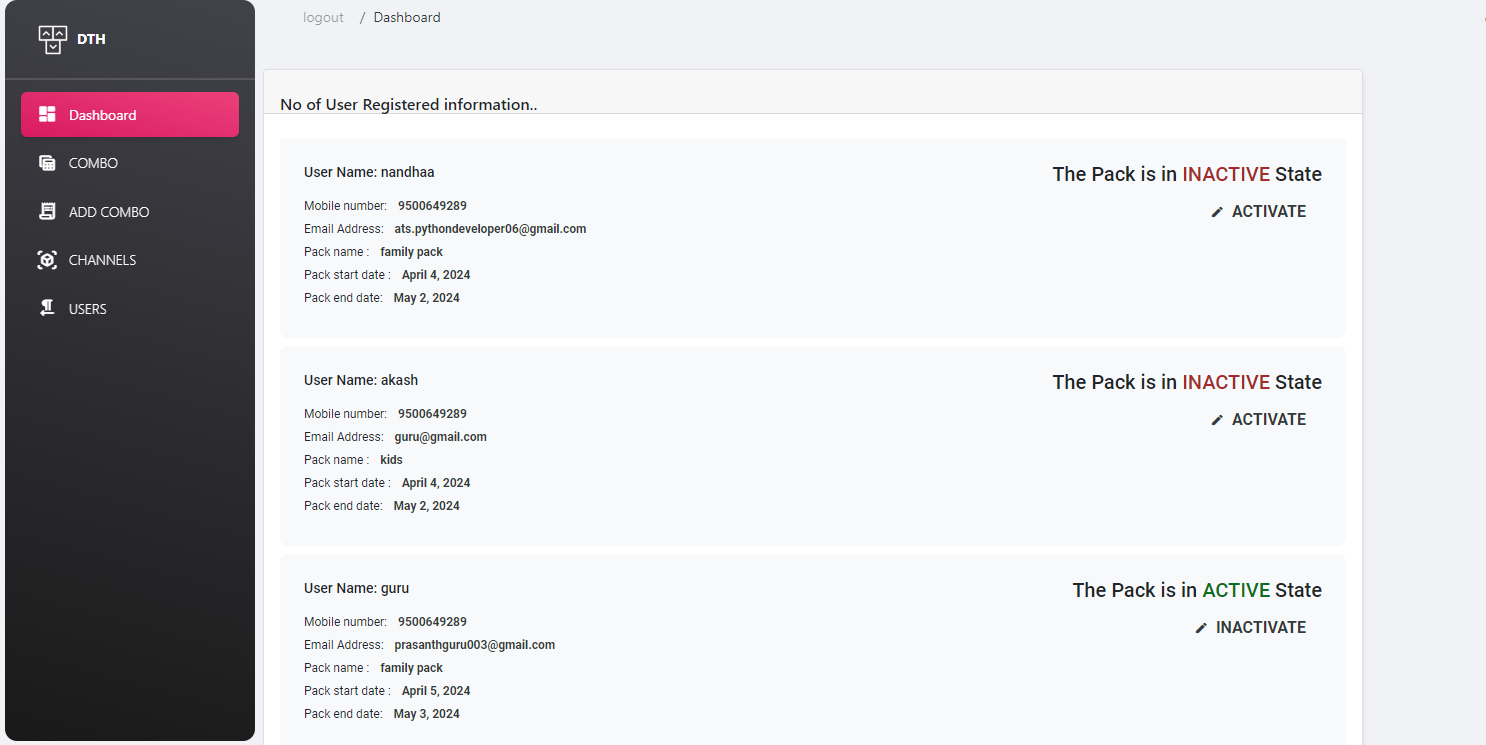
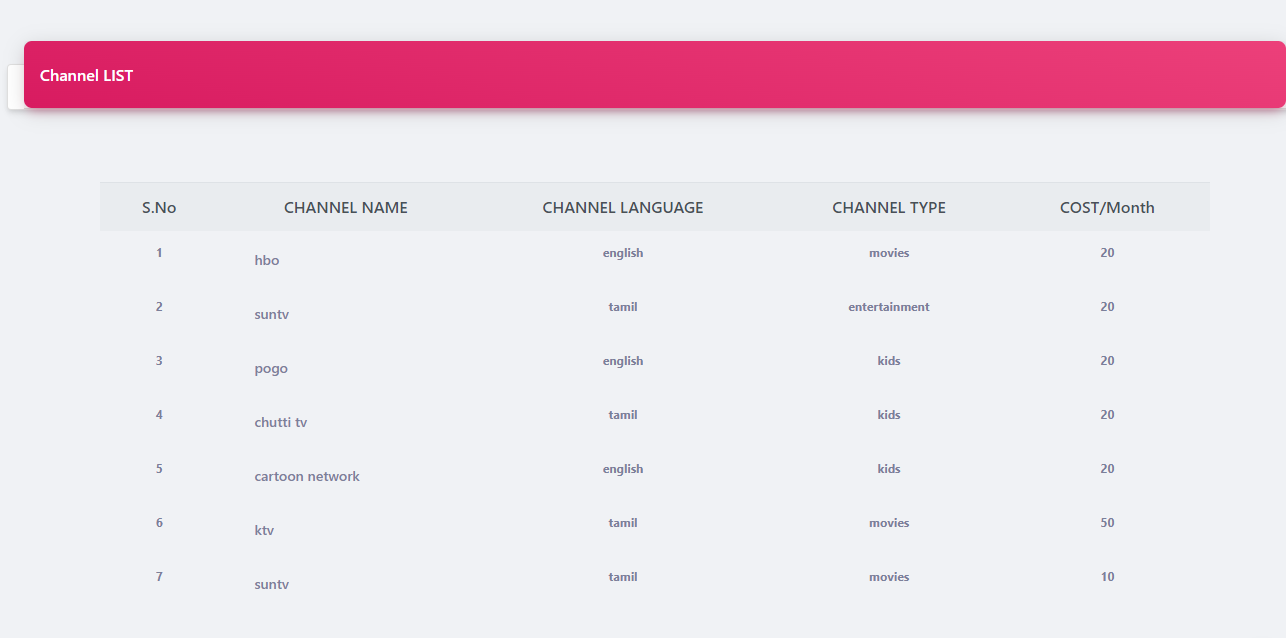
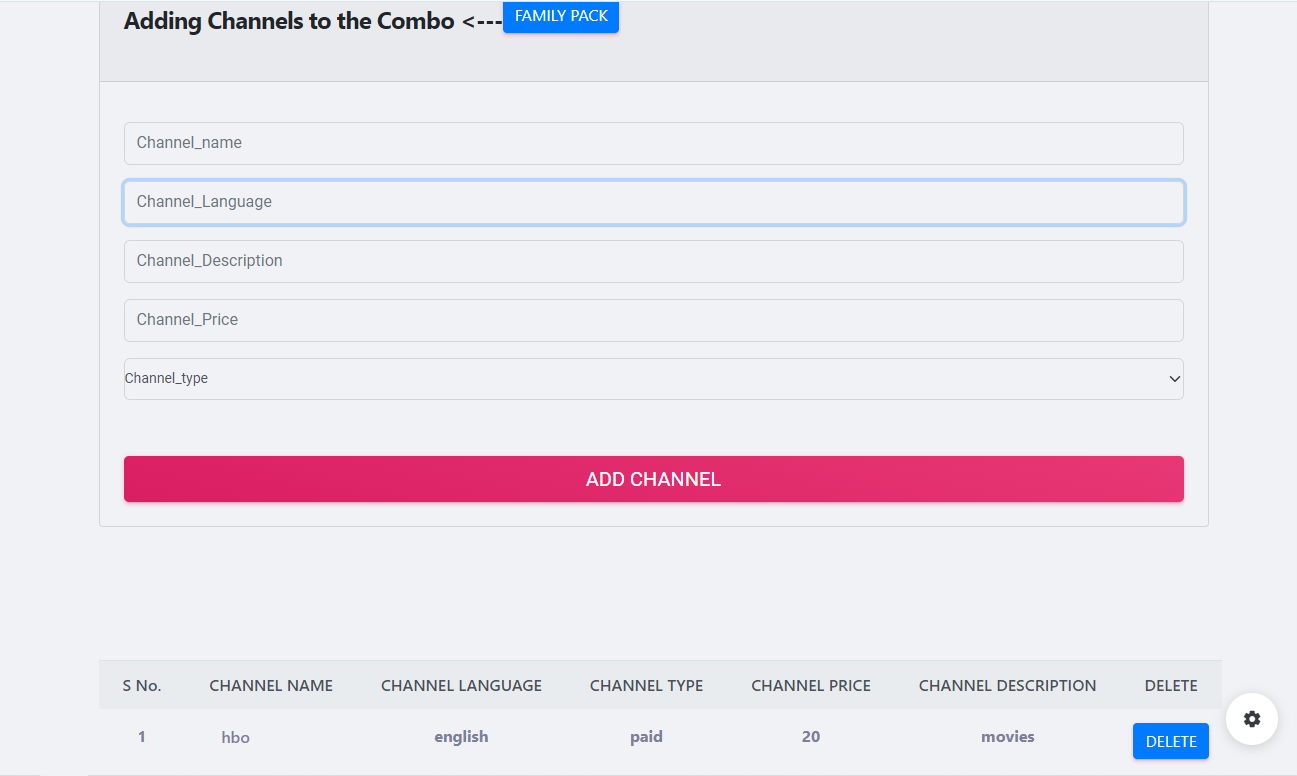
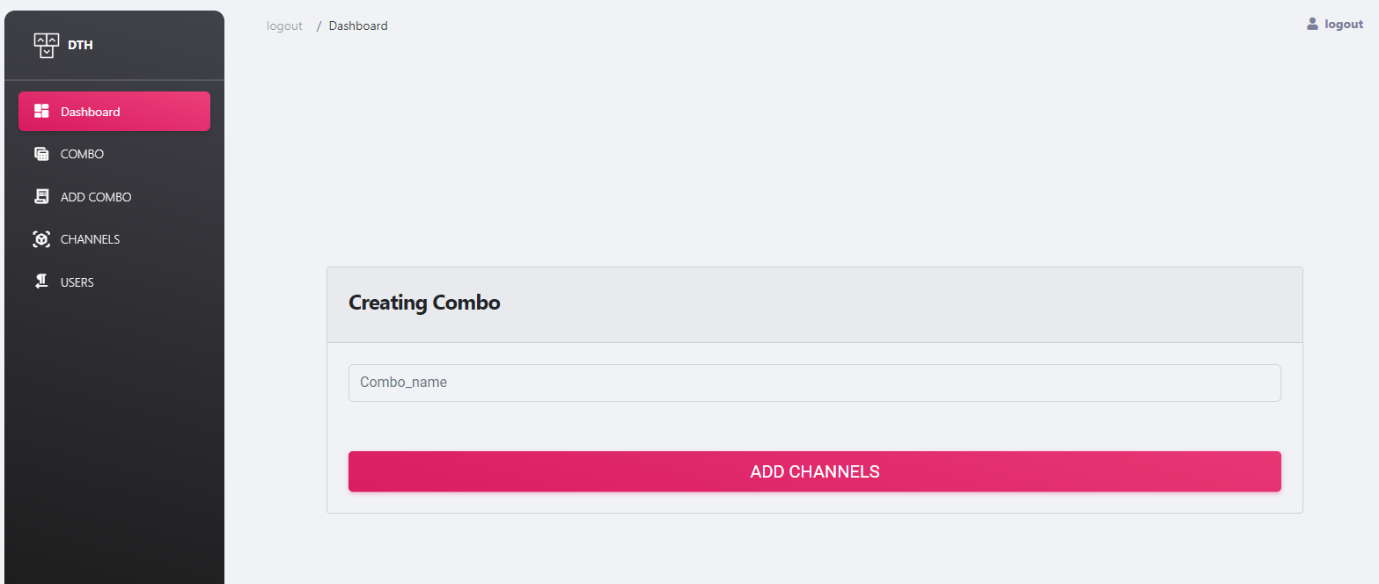
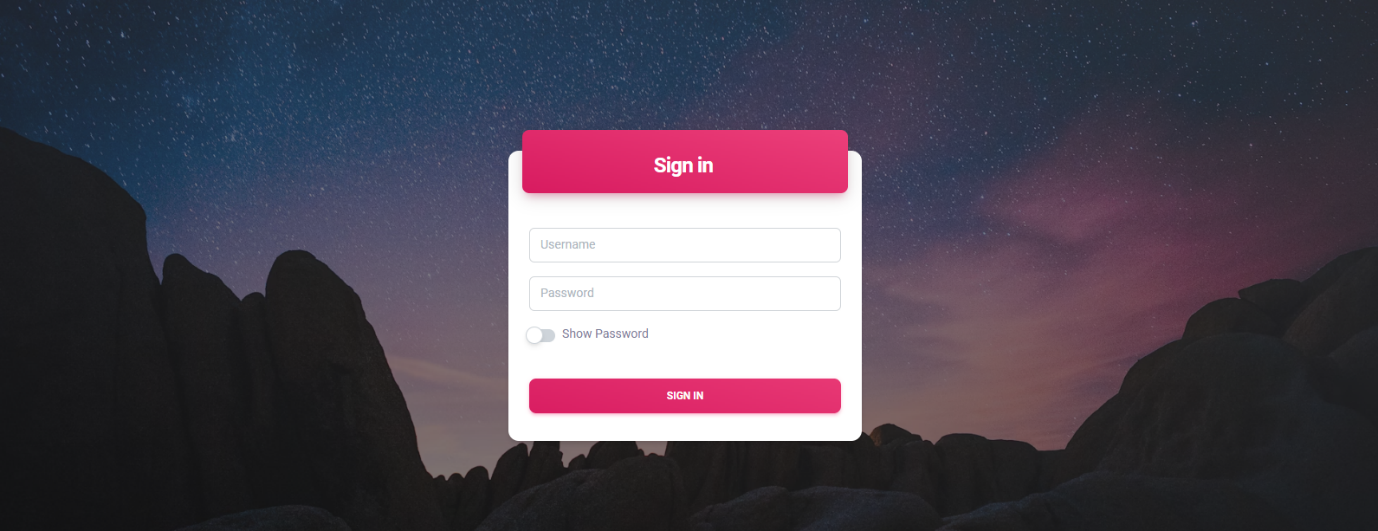
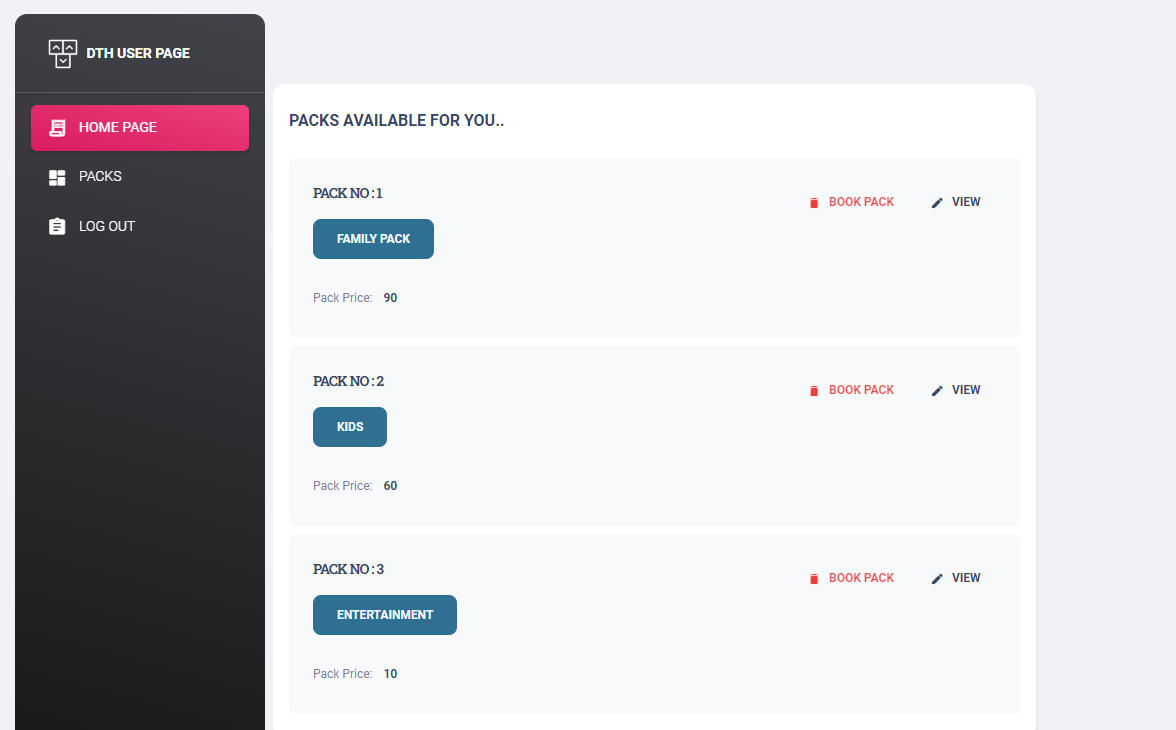
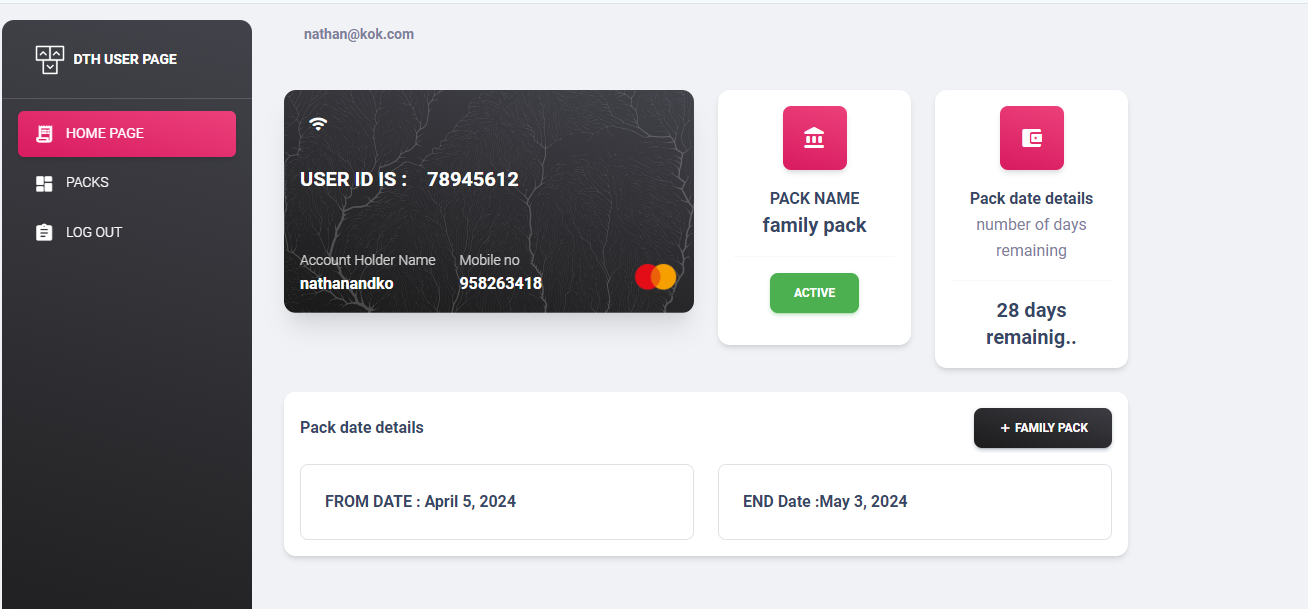
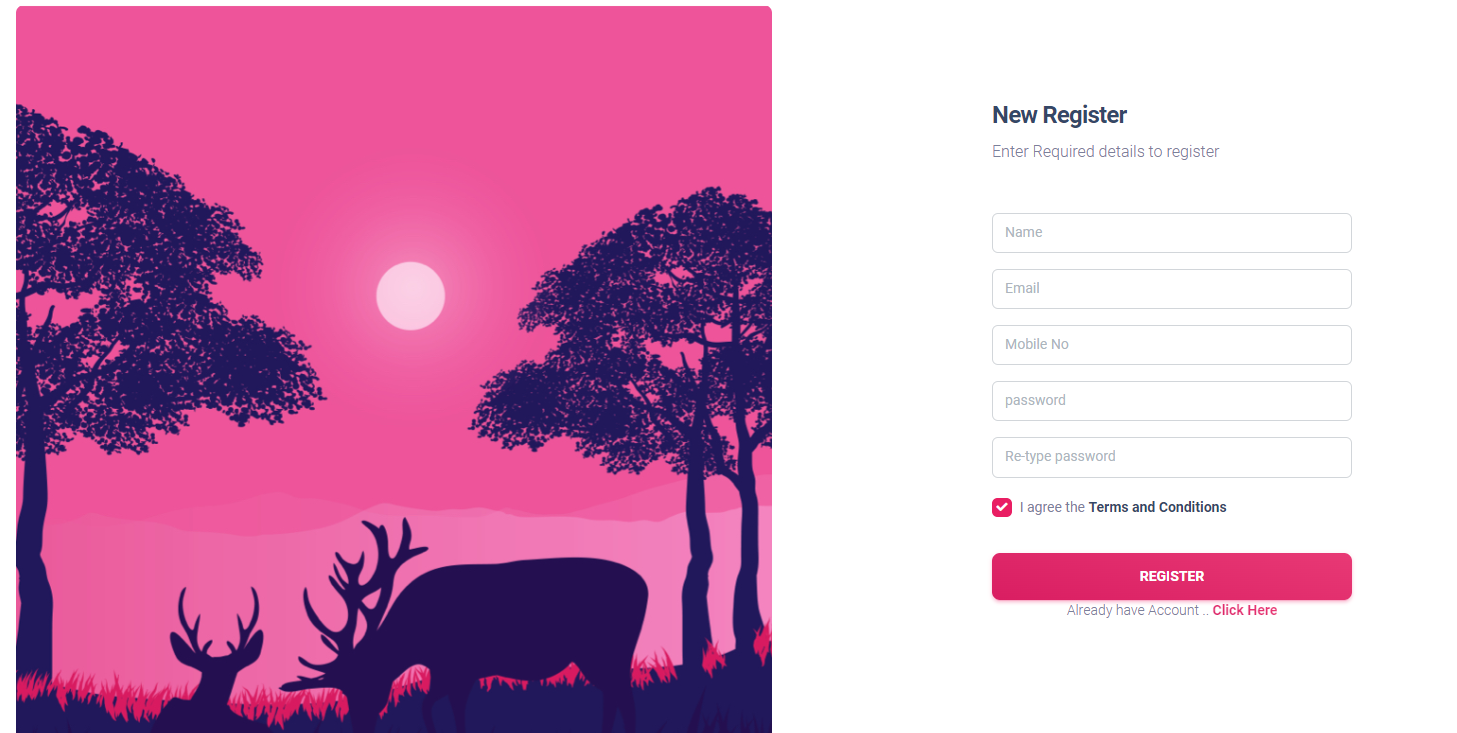
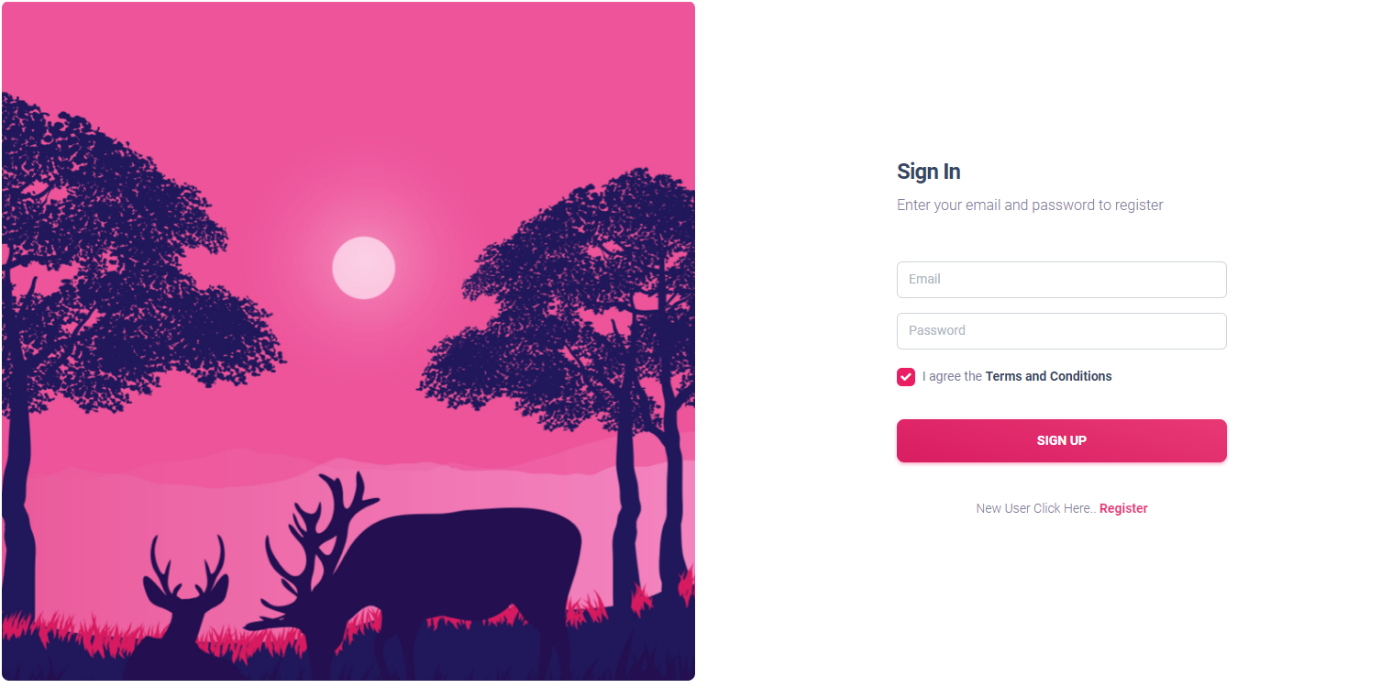
Books:

* Bouwman, Harry, et al. "Subscription Management and Billing for Pay-TV Services." Pay TV Business Models, Cambridge Scholars Publishing, 2016, pp. 85-102.
* Singhal, Mridul, and Sushil Kumar. "Direct-to-Home (DTH) Services: Subscriber Management and Billing." Satellite Broadcasting and Communication Systems, Springer, 2022, pp. 197-218.
* Choudhury, Jatindra N. "QR Code-Based Payment Systems." Mobile Payment Systems and Security, CRC Press, 2021, pp. 145-162.

Journal Articles:

* Gupta, Sanjeev, and Rajesh Sharma. "A Comprehensive Study of Subscriber Management System for Direct-to-Home Services." International Journal of Computer Applications, vol. 92, no. 7, 2014, pp. 19-24.
* Chaudhary, Sushil, et al. "QR Code-Based Payment System: A Secure and Efficient Way for Financial Transactions." International Journal of Advanced Research in Computer Science and Software Engineering, vol. 11, no. 5, 2021, pp. 408-413.
* Sharma, Gaurav, and Priya Gupta. "Role of Email Notifications in Enhancing Customer Engagement for DTH Services." Journal of Digital Marketing and Advertising, vol. 6, no. 3, 2019, pp. 221-229.

**SCREENSHOTS :**

****