CHAPTER 1

INTRODUCTION

1.1 Overview

The Library Management System (LMS) is a web-based application built using the Django framework, designed to streamline and enhance the operations of a library. This system offers a comprehensive solution for managing books, users, and transactions efficiently. Key administrative functionalities include the ability to add, update, and remove books from the library's inventory. Each book entry contains essential details such as the book ID, title, author, category, location within the library, and the number of copies available. Admins also have the capability to manage user information by registering new users, updating existing user details, and removing users as necessary. One of the crucial features is the book issuing process, where admins can issue books to registered users, automatically updating the inventory by reducing the available copies by one. When books are returned, the inventory is updated to reflect the increased availability.

For users, the LMS provides an intuitive interface that allows them to log in and access various features. Users can search for books using a dedicated search bar by entering keywords such as title, author, or category. They can view detailed information about available books, including the author, book ID, location, and the number of copies available. Additionally, users can view a list of books they have currently issued, which helps them manage their borrowed books and track due dates. An important aspect of the system is the fine calculation feature. If a user fails to return a book by its due date, a fine of 10 rupees per day is imposed. This fine is automatically calculated and reflected in the user's dashboard under the issued book section, providing clear visibility and accountability.

The system architecture of the LMS is based on the Model-View-Template (MVT) design pattern, which ensures a clear separation of concerns and makes the system modular and maintainable. Models represent the data structure, capturing details about books, user information, and issued book records. Views handle the business logic for processing user requests and updating data accordingly. Templates define the HTML structure and presentation of the web pages, ensuring a user-friendly interface. This architecture not only supports efficient data management and user interactions but also allows for scalability, enabling the

addition of new features and functionalities as the library's needs grow.

Overall, the Library Management System significantly improves the efficiency, accuracy, and accessibility of library management. By automating repetitive tasks such as inventory management and user registration, the system reduces the workload on library staff. It minimizes human error through systematic data handling and provides users with easy access to library resources. The fine calculation and tracking feature further enhances the system by ensuring timely returns and accountability, ultimately contributing to a more organized and user-friendly library experience.

1.2 Problem statement

The traditional manual library management system is inefficient and error-prone, increasing staff workload and reducing user accessibility. Tasks like managing inventories and tracking issued books are time-consuming and prone to inaccuracies. This project aims to develop a web-based Library Management System using Django to automate these processes, improve accuracy, and enhance user experience.

Issues with Manual Library Management System

1. **Inefficiency**: Manual processes are time-consuming and require significant effort from staff for tasks such as cataloging books, registering users, and issuing or returning books.

2. **Inaccuracy**: Human errors are common in manual record-keeping, leading to incorrect data entries, misplaced books, and unreliable inventory counts.

3. **Limited Accessibility**: Users often need to visit the library in person to search for books or check their borrowing status, making it inconvenient and time-consuming.

4. **Poor Tracking**: Keeping track of issued and returned books manually can lead to lost or unreturned books and difficulties in enforcing return deadlines and fines.

5. **Lack of Real-time Information**: Manual systems do not provide real-time updates on book availability or user account status, which can frustrate users and hinder effective library management.

6. **Inadequate Reporting**: Generating reports on library usage, book popularity, or user activity manually is cumbersome and less efficient compared to automated systems.

CHAPTER 2

PROPOSED METHOD

About project

The Library Management System (LMS) project aims to create a web-based application using the Django framework to streamline and automate library operations. This system will manage book inventories, user information, and transactions efficiently, providing real-time updates and an intuitive user interface.

2.1 Aim

To develop a comprehensive Library Management System that automates book and user management processes, improves accuracy, and enhances the user experience for both library staff and patrons.

2.2 Objectives

- To automate the addition, updating, and removal of books and users in the library database.
- To implement a user-friendly interface for searching and viewing book details and availability.
- To enable admins to issue and track the return of books, automatically updating the inventory.
- To integrate a fine calculation system for overdue books and display this information on the user dashboard.

CHAPTER 3

REQUIEREMENTS ANALYSIS

3.1 Software requirements:

Programming languages:

Front End:

- HTML
- CSS
- JAVASCRIPT
- Bootstrap

BackEnd:

- Python (Django)
- Sqlite 3

Operating system :

ANY OS (Recommended: Windows8, Windows XP)

Application required :

- Standalone desktop application,
- PIP(pip –version above 3.8.0) Django (version above 3.0) &
- mysqlclient Python IDE(VS Code) &
- DBbrowser (SQlite3).

3.2 Hardware requirements:

CPU :

Pentium IV 2.4 GHz or above

Memory (Primary) :

512 MB, 1 GB or above

Hard Disk :

40 GB, 80GB, 160GB or above

Monitor :

15 VGA color

CHAPTER 4

IMPLEMENTATION

4.1 Working Modules:

- **Administrators**: Responsible for overseeing system-wide operations, managing book inventories, and handling user roles and permissions. They ensure the smooth functioning of the LMS by adding, updating, and removing books and users, issuing books to users, and managing the return process, including updating inventory and calculating fines for overdue books.

- **Users**: Registered users are responsible for browsing the library catalog, searching for books by title, author, or category, and viewing book details such as location and availability. They can also track issued books, monitor due dates, and view fines for overdue books on their dashboard.

Applications

- **Library Management**: This project can be utilized by libraries to streamline the management of book inventories, user registrations, and book transactions, thereby improving operational efficiency and accuracy.

- **Administrator Panel**: The Admin Dashboard serves as a centralized hub for managing book inventories, user roles, permissions, and system settings. It facilitates the issuance and return of books and updates inventory in real-time, ensuring smooth operation and oversight of the library system.

- **User Dashboard**: The User Dashboard provides users with an intuitive interface to browse the library catalog, search for books, and view detailed information about book availability. It also allows users to track their issued books, due dates, and fines for overdue books, enhancing their library experience.

5.2 Code Snippets:

Models.py:

The models.py file in a Django project serves as the blueprint for defining the structure and behavior of the database tables used by the application. It encapsulates the data model layer of the MVC (Model-View-Controller) architecture, facilitating the interaction between the application's business logic and the underlying database.

Within **models.py,** developers define Python classes that represent database tables, with each class corresponding to a specific table in the database schema. These classes inherit from Django's **Model** class, which provides the functionality to interact with the database through object-oriented programming paradigms.

By defining the database schema in Python code within **models.py**, Django abstracts away the complexities of SQL (Structured Query Language) queries and database management. Instead, developers interact with the database using high-level Python objects and Django's built-in ORM (Object-Relational Mapping) system.

The ORM system translates Python code into SQL queries, allowing seamless communication between the application's Python codebase and the database backend. This abstraction layer simplifies database operations, enhances code readability, and promotes code reusability, facilitating rapid development and maintenance of Django applications.

```
from django.db import models
from django.contrib.auth.models import User
from django.utils import timezone


class Book(models.Model):
    title = models.CharField(max_length=255)
    isbn = models.CharField(max_length=13, unique=True)
    author = models.CharField(max_length=255)
    category = models.CharField(max_length=100)
    total_copies = models.PositiveIntegerField()
    available_copies = models.PositiveIntegerField()
    location = models.CharField(max_length=30, null=True)
```

```
    def __str__(self):
        return self.title

class IssuedBook(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    book = models.ForeignKey(Book, on_delete=models.CASCADE)
    isbn = models.CharField(max_length=13)
    issue_date = models.DateField(auto_now_add=True)
    due_date = models.DateField()
    is_returned = models.BooleanField(default=False)

    def __str__(self):
        return f"{self.book.title} issued to {self.user.username}"

    @property
    def fine_amount(self):
        if self.is_returned:
            return 0
        today = timezone.now().date()
        if today > self.due_date:
            overdue_days = (today - self.due_date).days
            return overdue_days * 10
        return 0


class users(models.Model):
    id = models.AutoField(primary_key=True)
    username = models.CharField(max_length=150)
    password = models.TextField(max_length=20)

    def __str__(self):
        return self.username
```

Views.py:

The **views.py** file in a Django project serves as the controller layer of the MVC (Model-View Controller) architecture, responsible for processing user requests, executing business logic, and rendering appropriate responses. It contains Python functions, known as view functions, that handle incoming HTTP requests and generate HTTP responses to be sent back to the client. Each view function typically corresponds to a specific URL endpoint defined in the project's URL configuration. When a user navigates to a particular URL, the corresponding view function is invoked to process the request and generate the appropriate response.

In Django, views are responsible for:

1. Request Processing: Views receive HTTP requests from clients and extract relevant data, such as request parameters, form data, or session information, to determine the desired action.

2. Business Logic Execution: Views execute the application's business logic, which may involve querying the database, performing calculations, or invoking external services to process the request and generate a response.

3. Response Rendering: Views generate HTTP responses to be sent back to the client, typically by rendering templates or returning JSON data. The response may include dynamic content generated based on the request data and application logic.

4. Template Rendering: In Django's typical usage, views often render HTML templates to generate the user interface presented to the client. Views pass data to templates, which are then rendered into HTML pages using Django's template engine.

5. Redirects and Error Handling: Views handle redirects to other URLs and handle errors or exceptions that occur during request processing, ensuring robustness and user friendly error messages.

Overall, the views.py file plays a crucial role in Django applications, serving as the bridge between the user interface, business logic, and data model layers. It encapsulates the application's functionality, orchestrating the flow of data and interactions between components

to deliver a cohesive and responsive user experience.

```
from datetime import timedelta
from django.shortcuts import render, redirect, get_object_or_404
from django.contrib.auth.decorators import login_required,  user_passes_test
from django.contrib.auth import authenticate, login as auth_login, logout
from django.contrib.auth.forms import UserCreationForm, AuthenticationForm
from django.utils import timezone
from .models import Book, IssuedBook
from django.contrib.auth.models import User

def is_admin(user):
    return user.is_superuser

@login_required(login_url='login')
@user_passes_test(is_admin)
def issue_book(request, book_id):
    book = get_object_or_404(Book, id=book_id)
    if request.method == 'POST':
        user_id = int(request.POST['user_id'])
        user = get_object_or_404(User, id=user_id)
        due_date = timezone.now().date() + timedelta(days=10)

        if book.available_copies > 0:
            IssuedBook.objects.create(user=user, book=book, isbn=book.isbn,
due_date=due_date)
            book.available_copies -= 1
            book.save()
            return redirect('admin_dashboard')
        else:
            return render(request, 'issue_book.html', {'book': book, 'error': 'No copies available'})

    users = User.objects.filter(is_superuser=False)
    return render(request, 'issue_book.html', {'book': book, 'users': users})
```

```
def signup(request):
    if request.method == "POST":
        form = UserCreationForm(request.POST)
        if form.is_valid():
            user = form.save()
            return redirect('login')
    else:
        form = UserCreationForm()
    return render(request, 'signup.html', {'form': form})

def login(request):
    if request.method == "POST":
        form = AuthenticationForm(data=request.POST)
        if form.is_valid():
            user = form.get_user()
            auth_login(request, user)
            return redirect('dashboard')
    else:
        form = AuthenticationForm()
    return render(request, 'login.html', {'form': form})

@login_required(login_url='login')
def signout(request):
    logout(request)
    return redirect('login')

@login_required(login_url='login')
def dashboard(request):
    if request.user.is_superuser:
        return redirect('admin_dashboard')
    return redirect('user_dashboard')

@login_required(login_url='login')
```

```
@user_passes_test(is_admin)
def admin_dashboard(request):
    books = Book.objects.all()
    issued_books = IssuedBook.objects.filter(is_returned=False)
    return render(request, 'admin_dashboard.html', {'books': books, 'issued_books':
issued_books})

@login_required(login_url='login')
def user_dashboard(request):
    issued_books = IssuedBook.objects.filter(user=request.user, is_returned=False)
    books = Book.objects.all()
    return render(request, 'user_dashboard.html', {'issued_books':
issued_books,'books':books})

@login_required(login_url='login')
@user_passes_test(is_admin)
def add_book(request):
    if request.method == 'POST':
        title = request.POST['title']
        isbn = request.POST['isbn']
        author = request.POST['author']
        category = request.POST['category']
        total_copies = int(request.POST['total_copies'])
        available_copies = total_copies
        location = request.POST['location']
        Book.objects.create(title=title, isbn=isbn, author=author, category=category,
total_copies=total_copies, available_copies=available_copies, location=location)
        return redirect('admin_dashboard')
    return render(request, 'add_book.html')

@login_required(login_url='login')
@user_passes_test(is_admin)
def return_book(request, issued_book_id):
    issued_book = get_object_or_404(IssuedBook, id=issued_book_id)
```

```
    issued_book.is_returned = True
    issued_book.save()
    book = issued_book.book
    book.available_copies += 1
    book.save()
    return redirect('admin_dashboard')

@login_required(login_url='login')

def search_books(request):
    if request.method == 'GET':
        query = request.GET.get('query')
        category = request.GET.get('category')
        books = Book.objects.all()
        if query:
            books = books.filter(title__icontains=query)
        if category:
            books = books.filter(category=category)
        return render(request, 'user_dashboard.html', {'books': books})
```

Admin.py

The admin.py file in a Django project is used to register models with the Django admin interface, enabling administrators to manage database records through a user-friendly web-based interface. It serves as the configuration layer for the Django admin site, allowing developers to customize the behavior and appearance of the admin interface for each model.

In **admin.py,** developers can perform the following tasks:

1. **Model Registration:** Developers can register models by importing them from the application's models.py file and using the **admin.site.register()** method to associate them with the Django admin interface. Once registered, administrators can perform CRUD (Create, Read, Update, Delete) operations on instances of these models directly from the admin site.

2. **Customizing Admin Options:** Developers can customize the appearance and behavior of model admin pages by defining subclasses of **admin.ModelAdmin** and specifying various options such as list display fields, search fields, list filters, fieldsets, and inline editing options. These customizations allow developers to tailor the admin interface to meet specific project requirements and improve usability.

3. **Inline Models:** Developers can define inline models within the admin interface to manage related model instances within the same admin page. This is useful for managing one-to-many or many-to-many relationships, as it provides a convenient way to edit related model instances without navigating to separate admin pages.

4. **Permissions and Security:** Developers can configure permissions and access control for models registered with the admin interface, restricting certain users' ability to view, add, change, or delete model instances based on their user roles and permissions. This helps enforce security policies and prevent unauthorized access to sensitive data.

Overall, the **admin.py** file plays a crucial role in Django applications, providing a powerful and customizable administrative interface for managing database records. By leveraging the features and capabilities of the Django admin site, developers can streamline administrative tasks, improve user productivity, and maintain data integrity within their applications.

```
from django.contrib import admin
from .models import Book, IssuedBook, users


class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'isbn', 'author', 'category', 'total_copies', 'available_copies')
    search_fields = ('title', 'author', 'isbn', 'category')
    list_filter = ('category',)


class IssuedBookAdmin(admin.ModelAdmin):
    list_display = ('book', 'user', 'issue_date', 'due_date', 'is_returned')
    search_fields = ('book__title', 'user__username', 'book__isbn')
    list_filter = ('is_returned', 'issue_date', 'due_date')
    autocomplete_fields = ['book', 'user']
```

```
class usersAdmin(admin.ModelAdmin):
    list_display = ('username', 'password')
    search_fields = ('username',)

admin.site.register(Book, BookAdmin)
admin.site.register(IssuedBook, IssuedBookAdmin)
admin.site.register(users, usersAdmin)
```

5 Conclusion:

In conclusion, the development of the Library Management System (LMS) marks a significant advancement in addressing the challenges associated with managing library operations. This system efficiently handles tasks such as book inventory management, user registration, book issuance, and return processes, providing a comprehensive solution for modern libraries. Through the collaborative efforts of our development team, we have successfully designed and implemented a robust platform that meets the needs of both library administrators and users.

The LMS offers a user-friendly interface and a range of features tailored to simplify library management. Administrators can easily manage book inventories, user roles, and permissions, ensuring the smooth operation of the library. Users benefit from intuitive search functionalities, detailed book information, and real-time updates on book availability and fines for overdue books. This system not only streamlines library processes but also enhances user experience by providing easy access to library resources and transparent tracking of borrowed books.

Overall, the Library Management System serves as a valuable tool for improving the efficiency and accuracy of library management, fostering better organization, and delivering an exceptional user experience. As we continue to iterate and enhance the system based on user feedback and emerging trends in library management, we remain committed to driving innovation and excellence in the field.

We extend our gratitude to all stakeholders, including our development team, project sponsors, testers, and end-users, for their invaluable contributions and support throughout the project lifecycle. With the successful deployment of the Library Management System, we look forward to empowering libraries and transforming the way library services are delivered and experienced in the digital age.