SaaS Task Manager: A Web Application for Task Management

Ruth

January 8, 2025

Abstract

The **Task Manager** web application is a Software as a Service (SaaS)[1] solution designed to help users efficiently manage their tasks. It allows users to create, view, update, and delete tasks, with features such as sorting tasks by various criteria like deadlines, status, and overdue tasks. The application is built using Flask[2] for the back-end, with PostgreSQL as the database, integrates Bootstrap[3] for responsive front-end design, and is deployed on Render[5] for reliable hosting. The project incorporates essential SaaS principles by offering user-friendly task management accessible from any device with an internet connection. Key features include basic user authentication, dynamic database interactions, and an intuitive interface, making it scalable and adaptable for diverse user needs. This project serves as a comprehensive learning experience in building SaaS applications, web development, database management, and user interface design. The application is deployed online, and can be accessed at: `https://saas-project-manager.onrender.com`.

Team

- **Author:** Ruth

1 Final Project Design

1.1 Design Overview

The application follows a client-server architecture where the front-end and back-end communicate through HTTP requests. Users can interact with the application through a web browser, while the back-end (Flask[2]) handles routing, data processing, and interaction with the database.

1.2 UML Diagrams

1.2.1 Use Case Diagram

This diagram will outline the various actors in the system (e.g., **Admin**, **User**) and their interactions with the system. It will depict common use cases such as "Login," "Register," "Create Task," "View Tasks," and "Logout."

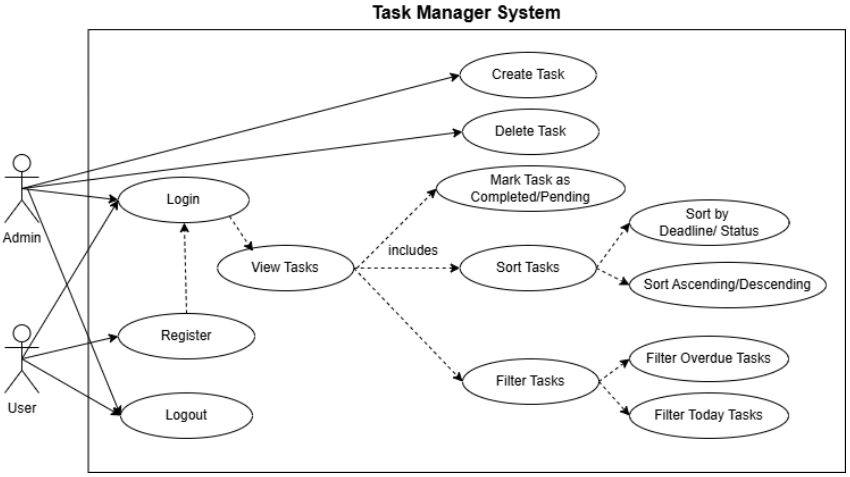

Figure 1: Use Case Diagram

1.2.2 Class Diagram

The class diagram will illustrate the main classes used in the application, such as **User**, **Task**, and **Database** (or **db**). This will show the relationships between these classes, their attributes, and methods.

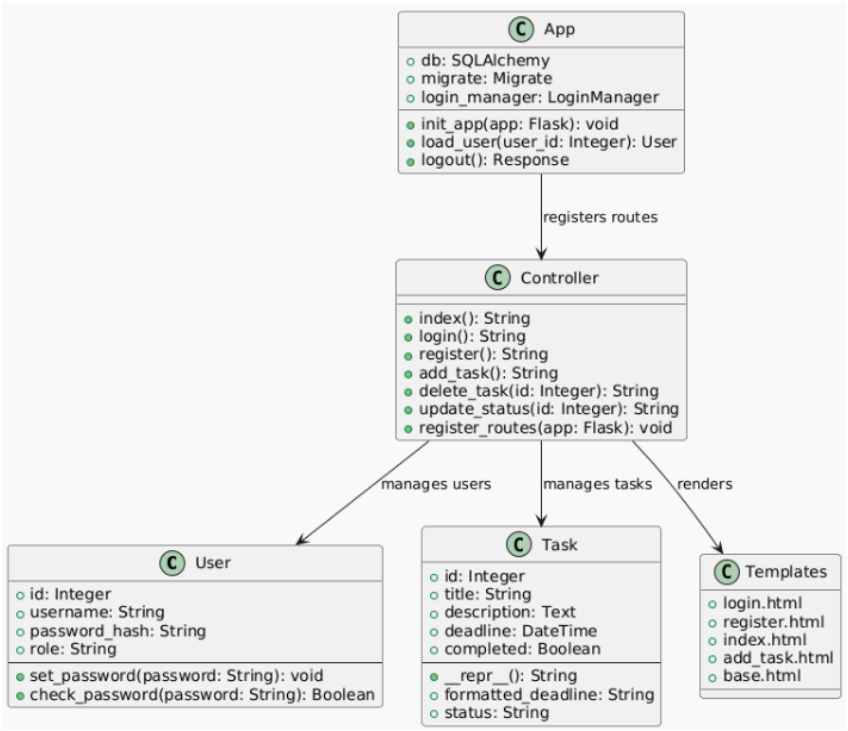

Figure 2: Class Diagram

1.2.3 Deployment Diagram

This diagram will show how the application is deployed on a server and how different components interact with each other, such as the web browser (client), web server (Flask[2]), and the database (PostgreSQL).

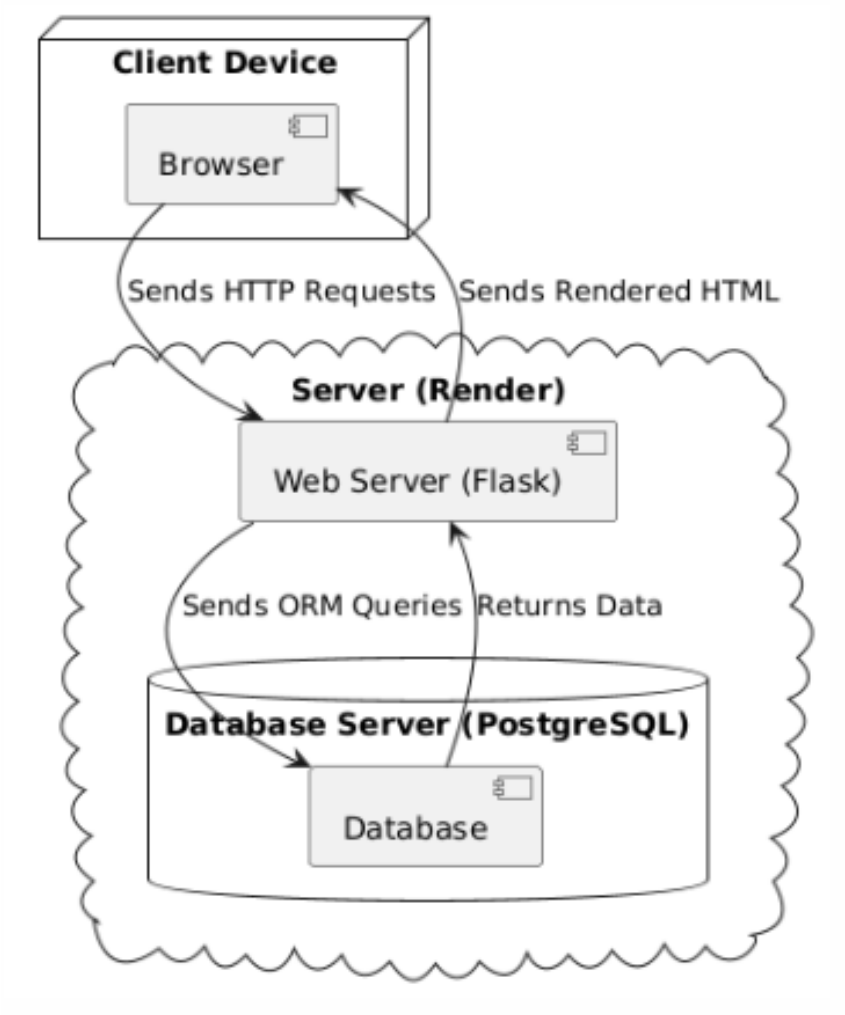

Figure 3: Deployment Diagram

2 Final Project Implementation

2.1 Source Code Overview

The project consists of multiple components:

- **Back-End (Flask[2])**:
  - `app.py`: Main entry point of the application that initializes the Flask[2] app and connects it to the database.
  - `controller.py`: Contains the route definitions for the application, such as user login, registration, and task management.
  - `model.py`: Defines the data models for tasks and users using SQLAlchemy[4].
  - `templates/`: Contains the HTML templates for the user interface, such as login, registration, task management pages, etc.
- **Database (PostgreSQL)**:
  - In `app.py`, PostgreSQL is connected using SQLAlchemy[4], and the connection URL is retrieved from the environment variable:

* `app.config['SQLALCHEMY_DATABASE_URI'] = os.getenv('DATABASE_URL')`

- **Front-End (HTML/CSS/JS)**:
  - `HTML`: The user interface is built with HTML, with dynamic pages rendered by Flask[2].
  - `CSS`: The application uses Bootstrap[3] for responsive design, ensuring a user-friendly interface across different devices.
  - `JavaScript`: For interactivity and enhancing the user experience, such as form validation, dynamic content loading, etc.
- **Deployment Platform**:
  - `Render`: The application is hosted on Render[5], a cloud service platform. It provides infrastructure for running the Flask[2] back-end and serves the PostgreSQL database.

2.2 Code Walkthrough

The complete code and project structure for this application are available on GitHub. You can access the repository at `https://github.com/ru2t7/Saas_project_manager`. This includes all files, such as the back-end code, front-end templates, and database configurations.

app.py: Initializes Flask[2] and configures the database connection. Registers routes and user authentication logic.

```
from flask import Flask, session, flash, redirect, url_for
from flask_session import Session
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from flask_login import LoginManager, login_required, logout_user


import os

# Initialize extensions
db = SQLAlchemy()  # Database instance
migrate = Migrate()  # Migration tool
login_manager = LoginManager()  # Login manager for handling user sessions

# Create the Flask app
app = Flask(__name__)
app.secret_key = os.environ.get('SECRET_KEY', 'fallback_secret_key')  # Secret key for session security

# Configure the app
app.config['SQLALCHEMY_DATABASE_URI'] = (
    'postgresql+pg8000://task_db_m4nd_user:40Ma9VCQ6jnyNqyGBXvvY1lMqxVi04k9'
    '@dpg-ctoo311opnds73fjflag-a.frankfurt-postgres.render.com/task_db_m4nd'
)  # Database connection URI
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False  # Disable modification tracking overhead
```

```
app.config['SESSION_COOKIE_SECURE'] = True  # Send cookies over HTTPS only
app.config['SESSION_COOKIE_HTTPONLY'] = True  # Prevent JavaScript access to cookies
app.config['SESSION_COOKIE_SAMESITE'] = 'Lax'  # Prevent cross-site request forgery

# Initialize extensions with the app
db.init_app(app)
migrate.init_app(app, db)
login_manager.init_app(app)
login_manager.login_view = 'login'  # Redirect unauthorized users to the login page

# Define user loader for Flask-Login
@login_manager.user_loader
def load_user(user_id):
    from models import User  # Import here to avoid circular import issues
    return User.query.get(int(user_id))  # Load user by ID from the database

# Configure session management
app.config['SESSION_TYPE'] = 'filesystem'  # Use server-side session storage
app.config['SESSION_PERMANENT'] = False  # Sessions are not permanent
app.config['SESSION_USE_SIGNER'] = True  # Sign session cookies to prevent tampering
Session(app)  # Initialize Flask-Session

# Logout route
@app.route('/logout', methods=['POST'])
@login_required
def logout():
    logout_user()  # Log out the current user
    flash('Logged out successfully.')  # Show a success message
    session.clear()  # Clear session data
    return redirect(url_for('login'))  # Redirect to the login page

# Register models and routes
with app.app_context():
    from models import User, Task  # Import models for database creation
    from routes import register_routes  # Import route registration function
    db.create_all()  # Ensure database tables are created
    register_routes(app)  # Register application routes

# Run the application
if __name__ == '__main__':
    app.run(debug=True)  # Start the Flask development server
```

Listing 1: app.py

controller.py: Defines the routes for the web pages, including `index`, `login`, `register`, `add_task`, etc. Handles dynamic data (tasks) by interacting with the database, restricts certain actions to admin users, and ensures secure interactions through login requirements.

```python
from flask import Flask, render_template, redirect, request, url_for, flash, session
from flask_login import login_user, logout_user, login_required, current_user
from sqlalchemy import func
from models import db, User, Task
from datetime import datetime


def register_routes(app):
    @app.route('/dashboard')
    def index():
        sort_by = request.args.get('sort_by', 'deadline')  # Default to 'deadline' if no sorting option is selected
        sort_direction = request.args.get('sort_direction', 'asc')  # Default direction to ascending if not specified

        # Get the current date for comparison
        today = datetime.utcnow().date()

        if sort_by == 'status':
            # Sorting tasks based on status priority: Overdue > Today > Pending > Completed
            case_condition = db.case(
                # Overdue: Tasks that have passed the deadline and are not completed
                ((Task.deadline < today) & (Task.completed == False), 1),
                # Due Today: Tasks whose deadline is today and not completed
                ((Task.deadline == today) & (Task.completed == False), 2),
                # Pending: Tasks that are not completed and are not overdue
                (Task.completed == False, 3),
                # Completed: Tasks that are marked as completed
                (Task.completed == True, 4),
                else_=5  # Default, catch-all for any tasks that don't match above
            )
            # Handle sorting direction (ascending or descending)
            if sort_direction == 'asc':
                tasks = Task.query.order_by(case_condition.asc(), Task.deadline.asc()).all()
            else:
                tasks = Task.query.order_by(case_condition.desc(), Task.deadline.desc()).all()

        elif sort_by == 'overdue':
            # Sorting tasks that are overdue (tasks with a past deadline and not completed)
            if sort_direction == 'asc':
                tasks = Task.query.filter(Task.deadline < datetime.utcnow(), Task.completed == False).order_by(
                    Task.deadline.asc()).all()
            else:
                tasks = Task.query.filter(Task.deadline < datetime.
```

```
                    utcnow(), Task.completed == False).order_by(
43                   Task.deadline.desc()).all()
44
45          elif sort_by == 'today':
46              # Sorting tasks that are due today
47              today = datetime.utcnow().date()
48              if sort_direction == 'asc':
49                  tasks = Task.query.filter(func.date(Task.deadline) ==
                        today, Task.completed == False).order_by(
50                       Task.deadline.asc()).all()
51              else:
52                  tasks = Task.query.filter(func.date(Task.deadline) ==
                        today, Task.completed == False).order_by(
53                       Task.deadline.desc()).all()
54
55          else:
56              # Default sorting by deadline (ascending or descending
                    based on direction)
57              if sort_direction == 'asc':
58                  tasks = Task.query.order_by(Task.deadline.asc()).all()
59              else:
60                  tasks = Task.query.order_by(Task.deadline.desc()).all()
61
62          return render_template('index.html', tasks=tasks, sort_by=
                sort_by, sort_direction=sort_direction)
63
64      @app.route('/', methods=['GET', 'POST'])
65      def login():
66          if current_user.is_authenticated:
67              # If the user is already logged in, redirect them to the
                    dashboard
68              return redirect(url_for('index'))  # Change 'dashboard' to
                    the name of your dashboard route
69
70          if request.method == 'POST':
71              username = request.form['username']
72              password = request.form['password']
73              user = User.query.filter_by(username=username).first()
74
75              if user and user.check_password(password):
76                  login_user(user)
77                  flash('Logged␣in␣successfully.')
78                  return redirect(url_for('index'))
79              else:
80                  flash('Invalid␣username␣or␣password.')
81
82          return render_template('login.html')
83
84      @app.route('/register', methods=['GET', 'POST'])
85      def register():
86          if request.method == 'POST':
87              username = request.form['username']
88              password = request.form['password']
89
90              if User.query.filter_by(username=username).first():
91                  flash('Username␣already␣exists.')
92                  return redirect(url_for('register'))
93
```

```
            new_user = User(username=username)
            new_user.set_password(password)
            db.session.add(new_user)
            db.session.commit()
            flash('User registered successfully.')
            return redirect(url_for('login'))

        return render_template('register.html')

    @app.route('/add', methods=['GET', 'POST'])
    @login_required
    def add_task():
        if current_user.role != 'admin':
            flash("You do not have permission to perform this action.",
                "error")
            return redirect(url_for('index'))

        if request.method == 'POST':
            title = request.form['title']
            description = request.form.get('description')
            deadline = datetime.strptime(request.form['deadline'], '%Y
               -%m-%d')
            new_task = Task(title=title, description=description,
               deadline=deadline)
            db.session.add(new_task)
            db.session.commit()
            return redirect(url_for('index'))

        return render_template('add_task.html')

    @app.route('/delete/<int:id>')
    @login_required
    def delete_task(id):
        if current_user.role != 'admin':
            flash("You do not have permission to perform this action.",
                "error")
            return redirect(url_for('index'))

        task = Task.query.get_or_404(id)
        db.session.delete(task)
        db.session.commit()
        return redirect(url_for('index'))

    @app.route('/update_status/<int:id>')
    @login_required
    def update_status(id):
        task = Task.query.get_or_404(id)
        task.completed = not task.completed
        db.session.commit()
        return redirect(url_for('index'))
```

Listing 2: controller.py

model.py: This code defines the two models for the task management system `User` which includes fields for username, hashed password, and role and methods to set and verify passwords, and `Task` with fields for title, optional description, deadline, and completed status. It provides formatted deadline output and calculates the task's status based on its deadline and completion state.

```python
from flask_sqlalchemy import SQLAlchemy
from werkzeug.security import generate_password_hash, check_password_hash
from flask_login import UserMixin
from datetime import datetime

# Use the db instance from app.py
from app import db

# User model
class User(db.Model, UserMixin):
    # Define table columns
    id = db.Column(db.Integer, primary_key=True)  # Primary key
    username = db.Column(db.String(150), unique=True, nullable=False)  # Unique username
    password_hash = db.Column(db.String(200), nullable=False)  # Hashed password for security
    role = db.Column(db.String(50), nullable=False, default='user')  # Role field with default value 'user'

    # Set hashed password
    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    # Check hashed password
    def check_password(self, password):
        return check_password_hash(self.password_hash, password)

# Task model
class Task(db.Model):
    # Define table columns
    id = db.Column(db.Integer, primary_key=True)  # Primary key
    title = db.Column(db.String(100), nullable=False)  # Task title
    description = db.Column(db.Text, nullable=True)  # Optional task description
    deadline = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)  # Deadline with default value
    completed = db.Column(db.Boolean, default=False)  # Completion status, default is False

    # String representation of a task instance
    def __repr__(self):
        return f"<Task {self.title}>"

    @property
    def formatted_deadline(self):
        """
        Format the deadline for display.
        Example: 'DD MMM YY' -> '25 Dec 23'.
        """
        return self.deadline.strftime('%d %b %y')

    @property
    def status(self):
        """
        Determine the task's status based on the deadline and completion status.
```

```
        Possible statuses:
        - "Completed": Task is marked as completed.
        - "Overdue": Deadline is past the current date.
        - "Due Today": Deadline matches the current date.
        - "Pending": Deadline is in the future and task is not
           completed.
        """
        today = datetime.utcnow().date()  # Current date
        deadline_date = self.deadline.date()  # Extract date from
           deadline
        if self.completed:
            return "Completed"
        elif deadline_date < today:
            return "Overdue"
        elif deadline_date == today:
            return "Due Today"
        else:
            return "Pending"
```

Listing 3: model.py

templates/base.html: This provides a template interface, it includes a Bootstrap[3]-based navigation bar with conditional links for login, register, and logout, depending on the user's authentication status. The content area is dynamic and customizable. Flash messages are displayed in a modal for user notifications, implemented with Bootstrap's JavaScript library.[3]

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale
       =1.0">
    <title>Task Manager</title>
    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/
       bootstrap.min.css" rel="stylesheet">
    <!-- Bootstrap Icons -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap-icons/font/
       bootstrap-icons.css" rel="stylesheet">
</head>

<body>
    <!-- Flash Message Modal -->
    <div class="modal fade" id="flashMessageModal" tabindex="-1" aria-
       labelledby="flashMessageModalLabel" aria-hidden="true">
        <div class="modal-dialog">
            <div class="modal-content">
                <div class="modal-header">
                    <h5 class="modal-title" id="flashMessageModalLabel"
                       >Notification</h5>
                    <!-- The 'X' button is already part of Bootstrap's
                       modal by default -->
                </div>
                <div class="modal-body" id="flashMessageContent">
                    <!-- Flash message content will be inserted
                       dynamically here -->
```

```
                </div>
                <!-- Footer omitted for simplicity, modal relies on the
                    default close button -->
            </div>
        </div>
    </div>

    <!-- Navigation Bar -->
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <div class="container-fluid">
            <a class="navbar-brand" href="#">Task Manager</a> <!--
                Brand name -->
            <div class="collapse navbar-collapse">
                <ul class="navbar-nav ms-auto"> <!-- Right-aligned
                    navigation items -->
                    {% if current_user.is_authenticated %}
                        <!-- Show Logout button if user is logged in --
                            >
                        <li class="nav-item">
                            <form action="{{ url_for('logout') }}"
                                method="POST">
                                <button type="submit" class="btn btn-
                                    danger">Logout</button>
                            </form>
                        </li>
                    {% else %}
                        <!-- Show Login button if user is not logged in
                            -->
                        <li class="nav-item">
                            <a href="{{ url_for('login') }}" class="btn
                                btn-primary">Login</a>
                        </li>
                        <!-- Show Register button if user is not logged
                            in -->
                        <li class="nav-item">
                            <a href="{{ url_for('register') }}" class="
                                btn btn-link">Register</a>
                        </li>
                    {% endif %}
                </ul>
            </div>
        </div>
    </nav>

    <!-- Content Block -->
    {% block content %}
    {% endblock %}

    <!-- Bootstrap JS Bundle with Popper -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/
        bootstrap.bundle.min.js"></script>

    <!-- Flash Messages Handling -->
    {% with messages = get_flashed_messages(with_categories=true) %}
        {% if messages %}
            <script>
                // Get the flash message content (first message)
                const flashMessageContent = '{{ messages[0][1] }}';
```

```
                const flashMessageModal = new bootstrap.Modal(document.
                    getElementById('flashMessageModal'));

                // Set the content of the modal
                document.getElementById('flashMessageContent').
                    textContent = flashMessageContent;

                // Show the modal
                flashMessageModal.show();
            </script>
        {% endif %}
    {% endwith %}
</body>

</html>
```

Listing 4: base.html

templates/add_task.html: This template is used for adding a new task. It extends the base template and contains a form with fields for the task title, description, and deadline. The form uses the POST method for submission, and the layout is styled with Bootstrap[3] classes. It includes a submit button for adding the task to the system.

```
{% extends "base.html" %} <!-- Inherit from the base HTML template -->

{% block content %} <!-- Define content for the "content" block in the
    base template -->
<div class="container my-5"> <!-- Main container with margin for
    spacing -->
    <h1 class="text-center">Add New Task</h1> <!-- Page heading
        centered -->

    <!-- Task form -->
    <form method="POST" class="mt-4"> <!-- Form submission uses POST
        method -->
        <!-- Task Title input -->
        <div class="mb-3"> <!-- Margin-bottom for spacing -->
            <label for="title" class="form-label">Task Title</label> <!
                -- Label for title -->
            <input type="text" name="title" id="title" class="form-
                control" required> <!-- Text input for task title -->
        </div>

        <!-- Task Description input -->
        <div class="mb-3">
            <label for="description" class="form-label">Description</
                label> <!-- Label for description -->
            <textarea name="description" id="description" class="form-
                control"></textarea> <!-- Textarea for optional task
                description -->
        </div>

        <!-- Task Deadline input -->
        <div class="mb-3">
            <label for="deadline" class="form-label">Deadline</label> <
                !-- Label for deadline -->
            <input type="date" name="deadline" id="deadline" class="
                form-control" required> <!-- Date picker for task
```

```
                deadline -->
        </div>

        <!-- Submit button -->
        <button type="submit" class="btn btn-success">Add Task</button>
            <!-- Submit form button -->
    </form>
</div>
{% endblock %} <!-- End of content block -->
```

Listing 5: add_task.html

templates/index.html: This template is used to display the task manager dashboard. It extends the base template and includes a sorting form to filter tasks by deadline, status, overdue, or today, with ascending or descending options. The tasks are displayed in a table with task details such as title, description, deadline, and status. Each task has options to mark it as completed or pending, and to delete it. The layout is styled using Bootstrap[3] classes for a responsive and user-friendly interface.

```
{% extends "base.html" %} <!-- Extends the base template -->

{% block content %}
<h1 class="text-center mb-4">Task Manager</h1> <!-- Page Title -->

<!-- Sorting Dropdown Form -->
<form method="GET" action="{{ url_for('index') }}" id="sortForm">
    <div class="form-group">
        <label for="sort_by">Sort Tasks By:</label>
        <div class="input-group">
            <!-- Dropdown for sorting criteria -->
            <select name="sort_by" id="sort_by" class="form-control"
                onchange="this.form.submit()">
                <option value="deadline" {% if request.args.get('
                    sort_by') == 'deadline' %}selected{% endif %}>
                    Deadline</option>
                <option value="status" {% if request.args.get('sort_by
                    ') == 'status' %}selected{% endif %}>Status</option>
                <option value="overdue" {% if request.args.get('sort_by
                    ') == 'overdue' %}selected{% endif %}>Overdue</
                    option>
                <option value="today" {% if request.args.get('sort_by')
                     == 'today' %}selected{% endif %}>Today</option>
            </select>

            <div class="input-group-append">
                <!-- Hidden input to toggle sort direction -->
                <input type="hidden" name="sort_direction" value="{{ '
                    asc' if sort_direction == 'desc' else 'desc' }}">

                <!-- Button to submit the form with sorting direction
                    -->
                <button type="submit" class="btn btn-outline-primary
                    sort-button">
                    <i class="bi {{ 'bi-arrow-up' if sort_direction ==
                        'asc' else 'bi-arrow-down' }}"></i>
                </button>
            </div>
        </div>
```

```
    </div>
</form>

<!-- Add New Task Button -->
<a href="/add" class="btn␣btn-primary␣mb-3">Add New Task</a>

<!-- Task Table -->
<div class="table-responsive">
    <table class="table␣table-striped">
        <thead>
            <tr>
                <th>#</th>
                <th>Title</th>
                <th>Description</th>
                <th>Deadline</th>
                <th>Status</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            {% for task in tasks %}
            <tr>
                <td>{{ task.id }}</td> <!-- Task ID -->
                <td>{{ task.title }}</td> <!-- Task Title -->
                <td>{{ task.description or "N/A" }}</td> <!-- Task
                    Description (N/A if empty) -->

                <td>
                    <!-- Display deadline with custom formatting -->
                    <div class="date-box">
                        <span class="day">{{ task.formatted_deadline.
                            split(' ')[0] }}</span>
                        <span class="month">{{ task.formatted_deadline.
                            split(' ')[1] }}</span>
                        <span class="year">{{ task.formatted_deadline.
                            split(' ')[2] }}</span>
                    </div>
                </td>

                <td>
                    <!-- Display task status with appropriate badge
                        color -->
                    <span class="badge
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{%␣if␣task.status␣==␣'Completed'␣%}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣bg-success
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{%␣elif␣task.status␣==␣'Overdue'␣%}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣bg-danger
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{%␣elif␣task.status␣==␣'Due␣Today'␣%}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣bg-warning
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{%␣else␣%}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣bg-primary
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{%␣endif␣%}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣">
                        {{ task.status }}
                    </span>
                </td>

                <td>
```

```
                    <!-- Actions: Update status or delete task -->
                    <a href="{{ url_for('update_status', id=task.id) }}
                       " class="btn btn-info btn-sm">
                        {% if task.completed %}
                            Mark as Pending
                        {% else %}
                            Mark as Completed
                        {% endif %}
                    </a>
                    <a href="/delete/{{ task.id }}" class="btn btn-
                       danger btn-sm">Delete</a>
                </td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
</div>
{% endblock %}
```

Listing 6: index.html

templates/login.html: This template extends the base layout and provides a login page with a centered form. The form includes fields for entering a username and password, with validation to ensure both are provided. If there are any error messages (e.g., invalid login), they are displayed in a Bootstrap[3] alert. The form also includes a link to the registration page for users who don't have an account, with the entire layout styled using Bootstrap[3]'s responsive grid and card components.

```
{% extends 'base.html' %} <!-- Extends the base layout -->

{% block content %}
<div class="container mt-5">
    <h2 class="text-center mb-4">Login</h2> <!-- Page Title -->

    <!-- Centered Login Form -->
    <div class="row justify-content-center">
        <div class="col-md-6">
            <div class="card shadow-sm">
                <div class="card-body">
                    <!-- Login Form -->
                    <form method="POST" action="{{ url_for('login') }}"
                       >
                        <!-- Username Field -->
                        <div class="mb-3">
                            <label for="username" class="form-label">
                               Username</label>
                            <input
                                type="text"
                                name="username"
                                id="username"
                                class="form-control"
                                placeholder="Enter your username"
                                required>
                        </div>

                        <!-- Password Field -->
                        <div class="mb-3">
                            <label for="password" class="form-label">
```

```
                                Password</label>
                            <input
                                type="password"
                                name="password"
                                id="password"
                                class="form-control"
                                placeholder="Enter your password"
                                required>
                        </div>

                        <!-- Error Message -->
                        {% if message %}
                        <div class="alert alert-danger" role="alert">
                            {{ message }} <!-- Displays error message
                                if present -->
                        </div>
                        {% endif %}

                        <!-- Submit Button -->
                        <div class="d-flex justify-content-between
                            align-items-center">
                            <button type="submit" class="btn btn-
                                primary w-100">Login</button>
                        </div>
                    </form>

                    <!-- Registration Link -->
                    <div class="text-center mt-3">
                        <a href="{{ url_for('register') }}" class="btn
                            btn-link">Don't have an account? Register</a
                            >
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Listing 7: login.html

templates/register.html: This template extends the base layout and provides a registration page with a centered form. The form includes fields for entering a username and password, with validation to ensure both fields are filled. Once submitted, the form sends the registration data via POST method. The layout uses Bootstrap[3]'s responsive grid system, and the registration button spans the full width of its container, providing a simple and user-friendly registration interface.

```
{% extends 'base.html' %} <!-- Extends the base layout -->

{% block content %}
<div class="container mt-5">
    <h2 class="mb-4 text-center">Register</h2> <!-- Page Title -->

    <!-- Centered Registration Form -->
    <div class="row justify-content-center">
        <div class="col-md-6">
            <form method="POST" action="{{ url_for('register') }}"> <!
```

```
                -- Form Submission -->

                <!-- Username Field -->
                <div class="mb-3">
                    <label for="username" class="form-label">Username</
                        label>
                    <input
                        type="text"
                        class="form-control"
                        id="username"
                        name="username"
                        required> <!-- Ensures username input is not
                            empty -->
                </div>

                <!-- Password Field -->
                <div class="mb-3">
                    <label for="password" class="form-label">Password</
                        label>
                    <input
                        type="password"
                        class="form-control"
                        id="password"
                        name="password"
                        required> <!-- Ensures password input is not
                            empty -->
                </div>

                <!-- Submit Button -->
                <button type="submit" class="btn␣btn-primary␣w-100">
                    Register</button> <!-- Full-width button -->
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

Listing 8: register.html

2.3 Project Features

- **User Authentication:** Allows users to register, log in, and log out.
- **Task Management:** Users can create tasks, set deadlines, and mark them as completed.
- **Task Sorting:** Tasks can be sorted by deadline, status (completed, pending), or other criteria.
- **Responsive UI:** Bootstrap[3] is used to make the UI responsive, ensuring it works well on both desktops and mobile devices.

3 References

References

[1] Armando Fox and David Patterson, *"Engineering Software as a Service: An Agile Approach Using Cloud"*, 2021. `https://saasbook.info/`

[2] Flask Documentation, `https://flask.palletsprojects.com/`

[3] Bootstrap Documentation, `https://getbootstrap.com/`

[4] SQLAlchemy Documentation, `https://docs.sqlalchemy.org/`

[5] Render Documentation, `https://render.com/docs`

4 Other Design or Implementation Models

4.1 Sorting Implementation

The task sorting feature was implemented using server-side logic with SQL queries to ensure efficient performance, even with large datasets. Tasks are sorted based on user-selected criteria such as deadlines, status, overdue tasks, or tasks due today. SQLAlchemy was used to construct and execute queries dynamically, enabling seamless integration with the PostgreSQL database. This approach minimizes client-side computation and ensures consistency in task ordering.

4.2 Error Handling

Custom error-handling mechanisms were implemented to improve the robustness and reliability of the application:

- **Validation Errors**: User inputs, such as empty task titles or invalid deadlines, are validated on both the client and server sides. Users are provided with immediate feedback using Bootstrap alert messages.
- **Database Connection Issues**: Failures in the database connection to PostgreSQL are caught, and user-friendly error messages are displayed instead of exposing raw stack traces.

4.3 Deployment Design

The application is deployed on **Render**, a cloud platform that simplifies deployment workflows for web applications and databases. Key aspects of the deployment include:

- **Database Configuration**: PostgreSQL is hosted on Render to provide a robust and scalable solution for data persistence. The application uses environment variables to securely store database credentials and connection strings.
- **Static Files and Security**: Static files, such as CSS and JavaScript, are efficiently served, and HTTPS is enforced for secure communication.

- **Scalability**: Render's auto-scaling feature ensures the app can handle increased traffic without manual intervention.

5 Conclusion

The Task Manager project successfully achieved its goal of providing a SaaS-based solution for managing tasks. The application allows users to create, update, delete, and sort tasks with intuitive functionality, offering a seamless user experience. Key features include user authentication, dynamic task sorting, and robust error handling.

Developing this project provided valuable insights into full-stack web development. The integration of Flask, PostgreSQL, and Bootstrap demonstrated the synergy of modern web technologies. The deployment on Render further emphasized the importance of scalable and reliable cloud-based platforms.

Future improvements to the application could include:

- Integrating task notifications via email or SMS to enhance user engagement.
- Adding support for recurring tasks to cater to more complex task management needs.
- Creating a User Management System with additional intermediary access levels to manage task permissions effectively.

This project has laid a strong foundation for building and scaling SaaS-based applications, combining efficient design and practical functionality.