Software Requirements Specification (SRS)

Task Management Application

Table of Contents

1. Introduction

1.1 Purpose

The purpose of this document is to define the requirements for the Task Management Application. This application is designed to help users manage their tasks efficiently by providing features such as task creation, updating, deletion, and reordering. The document will serve as a guide for developers, testers, and stakeholders throughout the project lifecycle.

1.2 Scope

The Task Management Application is a web-based system that allows users to:

- Register and log in securely.
- Create, update, delete, and view tasks.
- Search for tasks by title.
- Reorder tasks dynamically.

The system is built using the Django framework and is intended for personal or small-team use.

1.3 Definitions, Acronyms, and Abbreviations

- Django: A Python-based web framework.
- SRS: Software Requirements Specification.
- UI: User Interface.
- API: Application Programming Interface.
- SQLite: A lightweight database used for development.

1.4 References

- Django Documentation: https://docs.djangoproject.com/

- GitHub Repository: [Link to your repository]

1.5 Overview

The rest of this document is organized as follows:

- Section 2 provides an overall description of the system.

- Section 3 details the system's functional requirements.

- Section 4 describes the external interfaces.

- Section 5 outlines non-functional requirements.

- Section 6 covers other requirements such as database and legal considerations.

2. Overall Description

2.1 Product Perspective

The Task Management Application is a standalone web application built using Django. It interacts with a database to store user and task information. The system is designed to be simple, intuitive, and secure.

2.2 Product Features

- User authentication (login and registration).

- Task creation, updating, deletion, and viewing.

- Task search functionality.

- Task reordering functionality.

2.3 User Classes and Characteristics

- End Users: Individuals who will use the application to manage their tasks.

- Administrators: Developers or system admins responsible for maintaining the application.

2.4 Operating Environment

- Frontend: HTML, CSS, JavaScript.

- Backend: Django (Python).

- Database: SQLite (for development), PostgreSQL/MySQL (for production).

- Browser Compatibility: Chrome, Firefox, Safari, Edge.

2.5 Design and Implementation Constraints

- The application must be built using Django.

- The system must support user authentication and authorization.

- The database must be scalable for future enhancements.

2.6 Assumptions and Dependencies

- Users have access to a modern web browser.

- The application will be hosted on a server with internet access.

- Django and its dependencies are installed on the server.

3. System Features

3.1 User Authentication

- Login: Users can log in using their credentials. Authenticated users are redirected to the task list page.

- Registration: New users can register by providing a username and password. After registration, they are automatically logged in.

3.2 Task Management

- Task Creation: Users can create new tasks with a title, description, and completion status.

- Task Updating: Users can update the details of existing tasks.

- Task Deletion: Users can delete tasks. Only tasks owned by the logged-in user can be deleted.

- Task Viewing: Users can view a list of their tasks and the details of individual tasks.

3.3 Task Search

- Users can search for tasks by title using a search bar.

3.4 Task Reordering

- Users can reorder tasks dynamically. The new order is saved in the database.

4. External Interface Requirements

4.1 User Interfaces

- Login Page: A form for users to log in.
- Registration Page: A form for new users to register.
- Task List Page: Displays a list of tasks with options to create, update, delete, and search tasks.
- Task Detail Page: Displays detailed information about a specific task.

4.2 Hardware Interfaces

- The application runs on standard web servers and is accessible via web browsers.

4.3 Software Interfaces

- Django: The backend framework.
- SQLite/PostgreSQL/MySQL: The database system.
- HTML/CSS/JavaScript: The frontend technologies.

4.4 Communication Interfaces

- The application uses HTTP/HTTPS for communication between the client and server.

5. Non-Functional Requirements

5.1 Performance Requirements

- The system should handle up to 1,000 concurrent users.
- Page load times should not exceed 2 seconds.

5.2 Security Requirements

- User passwords must be hashed and stored securely.
- Only authenticated users can access task management features.

5.3 Usability Requirements

- The user interface should be intuitive and easy to navigate.
- The application should be accessible on both desktop and mobile devices.

5.4 Reliability Requirements

- The system should have an uptime of 99.9%.
- Data loss should be prevented through regular backups.

6. Other Requirements

6.1 Database Requirements

- The database should store the following tables:

  - Users: User credentials and information.

  - Tasks: Task details, including title, description, completion status, and user association.

6.2 Legal and Compliance Requirements

- The application must comply with GDPR for user data protection.

- All third-party libraries and frameworks must be properly licensed.

Conclusion

This SRS document provides a comprehensive overview of the Task Management Application's requirements. It serves as a foundation for the development, testing, and deployment of the system. Any changes to the requirements must be documented and approved by all stakeholders.

implimentation

Below is a step-by-step workflow to implement your Task Management Application using Django. This guide covers everything from setting up the environment to deploying the application.

Workflow to Implement the Task Management Application

Step 1: Set Up the Development Environment

1. Install Python:

- Download and install Python from [python.org](https://www.python.org/).
- Verify the installation:

```bash
python --version
```

2. Install Django:

- Install Django using pip:

```bash
```

pip install django
```

- Verify the installation:

```bash
django-admin --version
```

3. Set Up a Virtual Environment (Optional but Recommended):

- Create a virtual environment:

```bash
python -m venv venv
```

- Activate the virtual environment:

  - On Windows:

```bash
venv\Scripts\activate
```

  - On macOS/Linux:

```bash
source venv/bin/activate
```
```

Step 2: Create a Django Project

1. Create the Project:

   - Run the following command to create a new Django project:

     ```bash
     django-admin startproject taskmanager
     ```

   - Navigate into the project directory:

     ```bash
     cd taskmanager
     ```

2. Run the Development Server:

   - Start the server:

     ```bash
     python manage.py runserver
     ```

   - Open your browser and go to `http://127.0.0.1:8000/` to verify the project is running.

Step 3: Create a Django App

1. Create the App:

   - Run the following command to create a new app:

     ```bash
     python manage.py startapp base
     ```

2. Register the App:

   - Add `'base'` to the `INSTALLED_APPS` list in `taskmanager/settings.py`:

     ```python
     INSTALLED_APPS = [
         ...
         'base',
     ]
     ```

Step 4: Define Models

1. Create the `Task` Model:

   - Open `base/models.py` and define the `Task` model:

     ```python
     from django.db import models
     from django.contrib.auth.models import User
     ```

```
class Task(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE, null=True, blank=True)
    title = models.CharField(max_length=200)
    description = models.TextField(null=True, blank=True)
    complete = models.BooleanField(default=False)
    created = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title

    class Meta:
        ordering = ['complete']
```

2. Run Migrations:

- Create and apply migrations:

```bash
python manage.py makemigrations
python manage.py migrate
```

Step 5: Create Views

1. Define Views:

- Open `base/views.py` and define the views (e.g., `TaskList`, `TaskDetail`, `TaskCreate`, `TaskUpdate`, `DeleteView`, etc.) as provided in your code.

2. Add URLs:

- Create a `urls.py` file in the `base` app and define the URLs:

```python
from django.urls import path
from . import views

urlpatterns = [
    path('', views.TaskList.as_view(), name='tasks'),
    path('task/<int:pk>/', views.TaskDetail.as_view(), name='task'),
    path('task-create/', views.TaskCreate.as_view(), name='task-create'),
    path('task-update/<int:pk>/', views.TaskUpdate.as_view(), name='task-update'),
    path('task-delete/<int:pk>/', views.DeleteView.as_view(), name='task-delete'),
    path('task-reorder/', views.TaskReorder.as_view(), name='task-reorder'),
]
```

- Include the `base` URLs in the project's `urls.py`:

  ```python
  from django.urls import path, include

  urlpatterns = [
      path('admin/', admin.site.urls),
      path('', include('base.urls')),
  ]
  ```

Step 6: Create Templates

1. Create a `templates` Folder:
   - Inside the `base` app, create a folder named `templates/base`.

2. Create HTML Templates:
   - Create the following templates:
     - `login.html`: For the login page.
     - `register.html`: For the registration page.
     - `task_list.html`: For the task list page.
     - `task_detail.html`: For the task detail page.
     - `task_form.html`: For the task creation and update forms.

- `base.html`: A base template for other templates to extend.

3. Example Template (`task_list.html`):

```html
{% extends 'base/base.html' %}

{% block content %}
<h1>My Tasks</h1>
<a href="{% url 'task-create' %}">Add Task</a>
<form method="GET">
    <input type="text" name="search-area" value="{{ search_input }}">
    <input type="submit" value="Search">
</form>
<ul>
    {% for task in tasks %}
    <li>
        <a href="{% url 'task' task.id %}">{{ task.title }}</a>
        <a href="{% url 'task-update' task.id %}">Edit</a>
        <a href="{% url 'task-delete' task.id %}">Delete</a>
    </li>
    {% endfor %}
</ul>
{% endblock %}
```

```

Step 7: Implement User Authentication

1. Add Login and Registration Views:
   - Use the `CustomLoginView` and `RegisterPage` views provided in your code.

2. Update URLs:
   - Add URLs for login and registration in `base/urls.py`:
     ```python
     from django.contrib.auth.views import LoginView
     from .views import RegisterPage

     urlpatterns = [
         path('login/', CustomLoginView.as_view(), name='login'),
         path('register/', RegisterPage.as_view(), name='register'),
         ...
     ]
     ```
     ```

Step 8: Test the Application

1. Create a Superuser:

   - Run the following command to create an admin user:

     ```bash

     python manage.py createsuperuser

     ```

   - Access the admin panel at `http://127.0.0.1:8000/admin/`.

2. Test All Features:

   - Register a new user.

   - Log in and create, update, delete, and search tasks.

   - Verify task reordering functionality.

Step 9: Deploy the Application

1. Choose a Hosting Platform:

   - Popular options include Heroku, AWS, or PythonAnywhere.

2. Deploy to Heroku (Example):

   - Install the Heroku CLI and login:

     ```bash

     heroku login
     ```

```

- Create a new Heroku app:

```bash
heroku create
```

- Add a `Procfile` to the project root:

```
web: python manage.py runserver 0.0.0.0:$PORT
```

- Push the code to Heroku:

```bash
git push heroku main
```

- Run migrations on Heroku:

```bash
heroku run python manage.py migrate
```

Step 10: Maintain and Scale

1. Monitor Performance:
   - Use tools like Django Debug Toolbar or New Relic to monitor performance.
```

2. Add Features:

   - Implement additional features like task categories, due dates, or reminders.

3. Scale the Database:

   - Switch to a production-ready database like PostgreSQL.