**Структура проекта**

csharp

release\_management/

│

├── app/ # Папка с серверной частью

│ ├── \_\_init\_\_.py

│ ├── main.py # Главный файл FastAPI

│ ├── models.py # Модели данных

│ ├── database.py # Подключение к базе данных

│ ├── auth.py # Логика аутентификации

│

├── static/ # Папка для фронтенда (HTML, CSS, JS)

│ ├── index.html

│ ├── app.js

│ └── style.css

└── requirements.txt # Список зависимостей

**1. Установка зависимостей**

Создайте виртуальную среду и установите необходимые зависимости:

bash

pip install fastapi uvicorn sqlalchemy pydantic python-jose passlib

В файл requirements.txt добавьте:

fastapi

uvicorn

sqlalchemy

pydantic

python-jose

passlib

**2. Код для серверной части (FastAPI)**

**2.1. Модели данных (models.py)**

python

from sqlalchemy import Column, Integer, String, Date

from pydantic import BaseModel

from database import Base

# Модели для базы данных

class ReleaseStage(Base):

\_\_tablename\_\_ = "release\_stages"

id = Column(Integer, primary\_key=True, index=True)

name = Column(String, index=True)

description = Column(String)

start\_date = Column(Date)

end\_date = Column(Date)

responsible\_person = Column(String)

# Модели для передачи данных через API

class ReleaseStageCreate(BaseModel):

name: str

description: str

start\_date: str

end\_date: str

responsible\_person: str

class Config:

orm\_mode = True

**2.2. Подключение к базе данных (database.py)**

python

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

SQLALCHEMY\_DATABASE\_URL = "sqlite:///./test.db" # или PostgreSQL, если нужно

engine = create\_engine(SQLALCHEMY\_DATABASE\_URL, connect\_args={"check\_same\_thread": False})

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

**2.3. Логика аутентификации и авторизации (auth.py)**

python

from datetime import datetime, timedelta

from fastapi import Depends, HTTPException, status

from fastapi.security import OAuth2PasswordBearer

from jose import JWTError, jwt

from passlib.context import CryptContext

# Настройка секретного ключа и алгоритма

SECRET\_KEY = "secret\_key"

ALGORITHM = "HS256"

ACCESS\_TOKEN\_EXPIRE\_MINUTES = 30

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

pwd\_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def create\_access\_token(data: dict, expires\_delta: timedelta = timedelta(minutes=ACCESS\_TOKEN\_EXPIRE\_MINUTES)):

to\_encode = data.copy()

expire = datetime.utcnow() + expires\_delta

to\_encode.update({"exp": expire})

return jwt.encode(to\_encode, SECRET\_KEY, algorithm=ALGORITHM)

def verify\_token(token: str):

try:

payload = jwt.decode(token, SECRET\_KEY, algorithms=[ALGORITHM])

return payload["sub"]

except JWTError:

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Token is invalid")

def get\_current\_user(token: str = Depends(oauth2\_scheme)):

return verify\_token(token)

**2.4. Основной файл с API логикой (main.py)**

python

from fastapi import FastAPI, Depends, HTTPException, status

from sqlalchemy.orm import Session

from pydantic import BaseModel

from typing import List

from database import get\_db

from models import ReleaseStage, ReleaseStageCreate

from auth import get\_current\_user

app = FastAPI()

# Маршрут для создания нового этапа релиза

@app.post("/release\_stages/", response\_model=ReleaseStageCreate)

def create\_release\_stage(stage: ReleaseStageCreate, db: Session = Depends(get\_db), current\_user: str = Depends(get\_current\_user)):

db\_stage = ReleaseStage(\*\*stage.dict())

db.add(db\_stage)

db.commit()

db.refresh(db\_stage)

return db\_stage

# Маршрут для получения всех этапов релиза

@app.get("/release\_stages/", response\_model=List[ReleaseStageCreate])

def get\_all\_release\_stages(db: Session = Depends(get\_db)):

return db.query(ReleaseStage).all()

# Маршрут для получения токена

@app.post("/token/")

def login\_for\_access\_token():

# Пример возвращаемого токена (в реальности проверка на пароли и т.д.)

return {"access\_token": "dummy\_token", "token\_type": "bearer"}

**3. Код для фронтенда**

**3.1. HTML (static/index.html)**

html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Release Management</title>

<link rel="stylesheet" href="style.css">

</head>

<body>

<h1>Release Management Platform</h1>

<div id="releaseStagesContainer">

<h2>Release Stages</h2>

<ul id="releaseStagesList"></ul>

</div>

<div id="addReleaseStage">

<h2>Create Release Stage</h2>

<form id="releaseForm">

<input type="text" id="name" placeholder="Name" required />

<textarea id="description" placeholder="Description" required></textarea>

<input type="date" id="start\_date" required />

<input type="date" id="end\_date" required />

<input type="text" id="responsible\_person" placeholder="Responsible Person" required />

<button type="submit">Create</button>

</form>

</div>

<script src="app.js"></script>

</body>

</html>

**3.2. JavaScript (static/app.js)**

javascript

const API\_URL = "http://localhost:8000"; // URL вашего API

// Получение всех этапов релиза

async function fetchReleaseStages() {

const response = await fetch(`${API\_URL}/release\_stages/`);

const stages = await response.json();

const stagesList = document.getElementById("releaseStagesList");

stagesList.innerHTML = "";

stages.forEach(stage => {

const li = document.createElement("li");

li.textContent = `${stage.name} - ${stage.responsible\_person}`;

stagesList.appendChild(li);

});

}

// Создание нового этапа релиза

document.getElementById("releaseForm").addEventListener("submit", async (event) => {

event.preventDefault();

const newStage = {

name: document.getElementById("name").value,

description: document.getElementById("description").value,

start\_date: document.getElementById("start\_date").value,

end\_date: document.getElementById("end\_date").value,

responsible\_person: document.getElementById("responsible\_person").value,

};

const response = await fetch(`${API\_URL}/release\_stages/`, {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify(newStage),

});

if (response.ok) {

fetchReleaseStages(); // Обновить список

} else {

alert("Failed to create release stage.");

}

});

// Инициализация

fetchReleaseStages();

**3.3. CSS (static/style.css)**

css

body {

font-family: Arial, sans-serif;

}

#releaseStagesContainer {

margin-bottom: 20px;

}

#addReleaseStage {

margin-top: 20px;

}

input, textarea, button {

display: block;

margin-bottom: 10px;

}

**4. Запуск сервера**

Для запуска сервера используйте команду:

bash

uvicorn app.main:app --reload

Для добавления дополнительных функциональностей и улучшений в текущую платформу для управления релизным процессом мобильного приложения СБОЛ, можно интегрировать следующие возможности:

1. **Уведомления о новых релизах**.
2. **Интерфейс для администратора**.
3. **Реализация аутентификации и авторизации** для пользователей.
4. **Расширенные формы для ввода данных**.
5. **Система ролей для пользователей**.

**1. Уведомления о новых релизах**

Для уведомлений можно добавить WebSocket-соединение, которое будет отправлять уведомления пользователю, когда новый релиз добавлен.

**1.1. Реализация WebSocket в FastAPI (main.py)**

Добавим WebSocket сервер для отправки уведомлений при создании нового релиза.

python

from fastapi import WebSocket, WebSocketDisconnect

# Список подключенных WebSocket клиентов

clients = []

@app.websocket("/ws/")

async def websocket\_endpoint(websocket: WebSocket):

await websocket.accept()

clients.append(websocket)

try:

while True:

# Ожидаем сообщений от клиента

await websocket.receive\_text()

except WebSocketDisconnect:

clients.remove(websocket)

**1.2. Отправка уведомлений о новом релизе**

Когда добавляется новый релиз, отправим сообщение всем подключённым пользователям через WebSocket.

python

async def notify\_clients(message: str):

for client in clients:

await client.send\_text(message)

@app.post("/release\_stages/", response\_model=ReleaseStageCreate)

async def create\_release\_stage(stage: ReleaseStageCreate, db: Session = Depends(get\_db), current\_user: str = Depends(get\_current\_user)):

db\_stage = ReleaseStage(\*\*stage.dict())

db.add(db\_stage)

db.commit()

db.refresh(db\_stage)

# Отправка уведомления всем клиентам

await notify\_clients(f"New release stage created: {stage.name}")

return db\_stage

**1.3. Подключение к WebSocket на фронтенде (JavaScript)**

javascript

const socket = new WebSocket('ws://localhost:8000/ws/');

socket.onmessage = function(event) {

alert(event.data); // Отображение уведомления

};

**2. Интерфейс для администратора**

Для панели администратора можно добавить возможности управления пользователями и релизами, такие как:

* Просмотр всех релизов.
* Удаление или изменение этапов релиза.
* Просмотр логов системы.

**2.1. Страница администратора**

В index.html добавим раздел для администрирования:

html

<div id="adminPanel">

<h2>Admin Panel</h2>

<button onclick="fetchAllUsers()">View Users</button>

<div id="userList"></div>

</div>

**2.2. Логика администрирования в JavaScript**

javascript

async function fetchAllUsers() {

const response = await fetch(`${API\_URL}/users/`);

const users = await response.json();

const userList = document.getElementById("userList");

userList.innerHTML = "";

users.forEach(user => {

const div = document.createElement("div");

div.textContent = `User: ${user.username}`;

userList.appendChild(div);

});

}

**3. Реализация аутентификации и авторизации**

Добавим более сложную аутентификацию с использованием токенов и ролей пользователей для разграничения доступа.

**3.1. Модели пользователей (models.py)**

python

from sqlalchemy import Boolean

from pydantic import BaseModel

class User(Base):

\_\_tablename\_\_ = "users"

id = Column(Integer, primary\_key=True, index=True)

username = Column(String, unique=True, index=True)

password = Column(String)

is\_active = Column(Boolean, default=True)

role = Column(String, default="user") # Роль: user, admin и т.д.

class UserCreate(BaseModel):

username: str

password: str

role: str

class Config:

orm\_mode = True

**3.2. Аутентификация с токенами**

Внесем изменения в файл auth.py, чтобы обрабатывать создание пользователей и проверку ролей:

python

from fastapi import HTTPException, status

from passlib.context import CryptContext

from sqlalchemy.orm import Session

from models import User, UserCreate

from database import get\_db

from jose import JWTError, jwt

def create\_user(user: UserCreate, db: Session):

hashed\_password = pwd\_context.hash(user.password)

db\_user = User(username=user.username, password=hashed\_password, role=user.role)

db.add(db\_user)

db.commit()

db.refresh(db\_user)

return db\_user

def verify\_role(token: str, required\_role: str, db: Session):

user\_id = verify\_token(token)

db\_user = db.query(User).filter(User.id == user\_id).first()

if db\_user.role != required\_role:

raise HTTPException(status\_code=status.HTTP\_403\_FORBIDDEN, detail="You don't have access")

return db\_user

**3.3. Обработка ролей в маршрутах**

python

@app.post("/users/")

def create\_new\_user(user: UserCreate, db: Session = Depends(get\_db)):

create\_user(user, db)

return {"message": "User created successfully"}

@app.get("/admin/approve\_release/{release\_id}")

def approve\_release(release\_id: int, db: Session = Depends(get\_db), token: str = Depends(oauth2\_scheme)):

verify\_role(token, "admin", db) # Проверяем роль пользователя

# Логика утверждения релиза

return {"message": f"Release {release\_id} approved"}

**4. Расширенные формы для ввода данных**

Добавим более сложные формы для ввода данных о релизах, например, с выбором из выпадающего списка или другими элементами.

html

<form id="releaseForm">

<input type="text" id="name" placeholder="Name" required />

<textarea id="description" placeholder="Description" required></textarea>

<input type="date" id="start\_date" required />

<input type="date" id="end\_date" required />

<select id="responsible\_person">

<option value="user1">User 1</option>

<option value="user2">User 2</option>

</select>

<button type="submit">Create</button>

</form>

**Создание моделей базы данных**

Вам нужно создать модели для SQLAlchemy, которые будут определять структуру таблиц в вашей базе данных.

Пример:

python

# models.py

from sqlalchemy import Column, Integer, String, Date, ForeignKey

from sqlalchemy.orm import relationship

from database import Base

class ReleaseStage(Base):

\_\_tablename\_\_ = 'release\_stages'

id = Column(Integer, primary\_key=True, index=True)

name = Column(String, index=True)

description = Column(String)

start\_date = Column(Date)

end\_date = Column(Date)

responsible\_person\_id = Column(Integer, ForeignKey('users.id'))

responsible\_person = relationship("User", back\_populates="release\_stages")

class User(Base):

\_\_tablename\_\_ = 'users'

id = Column(Integer, primary\_key=True, index=True)

username = Column(String, unique=True, index=True)

password = Column(String)

role = Column(String, default="user")

release\_stages = relationship("ReleaseStage", back\_populates="responsible\_person")

Теперь создайте файл для создания базы данных и ее таблиц:

python

# database.py

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

DATABASE\_URL = "sqlite:///./test.db" # Используем SQLite для теста, в реальной ситуации используйте PostgreSQL

engine = create\_engine(DATABASE\_URL, connect\_args={"check\_same\_thread": False})

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

# Функция для создания таблиц

def init\_db():

Base.metadata.create\_all(bind=engine)

Теперь напишем скрипт, который будет заполнять базу данных начальными тестовыми данными. Для этого можно использовать SessionLocal для взаимодействия с БД.

python

# create\_db.py

from sqlalchemy.orm import Session

from models import User, ReleaseStage

from database import SessionLocal, init\_db

def create\_initial\_data(db: Session):

# Добавление пользователей

user1 = User(username="admin", password="hashed\_password", role="admin")

user2 = User(username="developer", password="hashed\_password", role="user")

db.add(user1)

db.add(user2)

db.commit()

db.refresh(user1)

db.refresh(user2)

# Добавление этапов релиза

release\_stage1 = ReleaseStage(

name="Stage 1: Planning",

description="Initial stage of planning.",

start\_date="2025-01-01",

end\_date="2025-01-10",

responsible\_person\_id=user1.id

)

release\_stage2 = ReleaseStage(

name="Stage 2: Development",

description="Development phase of the release.",

start\_date="2025-01-11",

end\_date="2025-02-01",

responsible\_person\_id=user2.id

)

db.add(release\_stage1)

db.add(release\_stage2)

db.commit()

# Инициализация БД и добавление данных

def init():

init\_db() # Создание таблиц

db = SessionLocal() # Открытие сессии

create\_initial\_data(db) # Добавление тестовых данных

db.close()

if \_\_name\_\_ == "\_\_main\_\_":

init() # Запуск процесса

Чтобы запустить процесс, нужно просто выполнить файл create\_db.py. Это создаст базу данных и заполнит её тестовыми данными:

bash

python create\_db.py

После этого будет создана база данных test.db (или другая, в зависимости от вашего настроенного пути), с таблицами пользователей и этапов релиза.

Подключение к базе данных в FastAPI

Для этого создайте зависимости для сессии с базой данных:

python

# main.py

from fastapi import FastAPI, Depends

from sqlalchemy.orm import Session

from database import SessionLocal

app = FastAPI()

# Функция для получения сессии с БД

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/release\_stages/")

def read\_release\_stages(db: Session = Depends(get\_db)):

release\_stages = db.query(ReleaseStage).all()

return release\_stages