College Branch Website Implementation Guide

Project Setup (Week 1)

Step 1: Environment Setup

1. **Install required tools**
   - Node.js and npm for React development
   - Python 3.9+ for FastAPI
   - PostgreSQL database
   - Git for version control

2. **Initialize project structure**

```
college-branch-website/
├── frontend/           # React frontend
├── backend/            # FastAPI backend
├── database/           # Database scripts
└── docs/               # Documentation
```

3. **Set up version control**

bash

```
git init
# Create .gitignore file for node_modules, env files, etc.
```

Step 2: Create Frontend Project

1. **Initialize React app**

bash

```
npx create-react-app frontend
cd frontend
```

2. **Install necessary packages**

bash

```
npm install react-router-dom axios formik yup jwt-decode @emotion/react @mui/materi
```

3. **Set up project structure**

```
frontend/
├── public/
├── src/
│   ├── components/     # Reusable components
│   ├── pages/          # Main page components
│   ├── services/       # API services
│   ├── context/        # React context for state
│   ├── utils/          # Utility functions
│   ├── assets/         # Images, styles, etc.
│   ├── App.js          # Main app component
│   └── index.js        # Entry point
```

Step 3: Create Backend Project

1. **Set up Python virtual environment**

bash

```
cd backend
python -m venv venv
# On Windows: venv\Scripts\activate
# On Unix: source venv/bin/activate
```

2. **Install dependencies**

bash

```
pip install fastapi uvicorn sqlalchemy psycopg2-binary pydantic python-jose[cryptog
```

3. **Create project structure**

```
backend/
├── app/
│   ├── api/
│   │   ├── endpoints/   # API route handlers
│   │   └── deps.py      # Dependencies
│   ├── core/            # Core functionality
│   │   ├── config.py    # Configuration
│   │   └── security.py  # Security utils
│   ├── db/              # Database
│   │   ├── base.py
│   │   └── session.py
│   ├── models/          # SQLAlchemy models
│   └── schemas/         # Pydantic schemas
├── alembic/             # Database migrations
└── main.py              # Entry point
```

Step 4: Set up Database

1. **Create PostgreSQL database**

bash

```
# Access PostgreSQL
psql -U postgres
# Create database
CREATE DATABASE college_branch_db;
# Create user (optional)
CREATE USER college_admin WITH PASSWORD 'secure_password';
GRANT ALL PRIVILEGES ON DATABASE college_branch_db TO college_admin;
```

2. **Initialize database migrations with Alembic**

bash

```
cd backend
alembic init alembic
# Configure alembic.ini and env.py
```

Database Schema Design (Week 2)

Step 1: Design Database Schema

1. **Users Table**

sql

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    hashed_password VARCHAR(100) NOT NULL,
    full_name VARCHAR(100),
    role VARCHAR(20) NOT NULL,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

2. **Students Table**

sql

```sql
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    roll_number VARCHAR(20) UNIQUE NOT NULL,
    full_name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(20),
    batch VARCHAR(10) NOT NULL,
    admission_year INTEGER NOT NULL,
    current_semester INTEGER,
    cgpa DECIMAL(4,2),
    profile_picture VARCHAR(255),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

3. **Achievements Table**

sql

```sql
CREATE TABLE achievements (
    id SERIAL PRIMARY KEY,
    student_id INTEGER REFERENCES students(id),
    title VARCHAR(100) NOT NULL,
    description TEXT,
    achievement_type VARCHAR(50),
    achievement_date DATE,
    certificate_url VARCHAR(255),
    is_verified BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

4. **Student Performance Table**

```sql
CREATE TABLE student_performance (
    id SERIAL PRIMARY KEY,
    student_id INTEGER REFERENCES students(id),
    semester INTEGER NOT NULL,
    sgpa DECIMAL(4,2),
    attendance_percentage DECIMAL(5,2),
    backlogs INTEGER DEFAULT 0,
    internship_status VARCHAR(50),
    remarks TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

5. **Settings Table**

```sql
CREATE TABLE settings (
    id SERIAL PRIMARY KEY,
    setting_key VARCHAR(50) UNIQUE NOT NULL,
    setting_value TEXT,
    description TEXT,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Step 2: Create SQLAlchemy Models

Create Python models that correspond to your database tables in `backend/app/models/`.

Backend Development (Weeks 3-4)

Step 1: Implement Core Functionality

1. **Configuration Setup (`app/core/config.py`)**
   - Database connection settings
   - JWT settings
   - Superset configuration
   - CORS settings
2. **Security Utilities (`app/core/security.py`)**
   - Password hashing
   - JWT token generation
   - Authentication logic
3. **Database Connection (`app/db/session.py`)**

- SQLAlchemy engine and session setup

Step 2: Create API Endpoints

1. **Authentication Endpoints**
   - Login/logout endpoints
   - User registration (for admins)
   - Password reset
2. **Student Endpoints**
   - CRUD operations for students
   - CSV import functionality
   - Search and filter students
3. **Achievement Endpoints**
   - CRUD operations for achievements
   - File upload for certificates
   - Google Forms integration
4. **Dashboard Endpoints**
   - Superset guest token generation
   - Dashboard configuration
5. **Settings Endpoints**
   - Get/update application settings

Step 3: Implement Superset Integration

1. **Set up Apache Superset**
   - Install and configure Superset
   - Connect Superset to your PostgreSQL database
   - Create initial dashboards
2. **Implement Guest Token Generation**

```python
from fastapi import APIRouter, Depends, HTTPException
import requests
from app.core.config import settings
from app.api.deps import get_current_user

router = APIRouter()

@router.get("/guest-token/{dashboard_id}")
def get_guest_token(
    dashboard_id: str,
    current_user = Depends(get_current_user)
):
    # Authenticate with Superset
    auth_response = requests.post(
        f"{settings.SUPERSET_URL}/api/v1/security/login",
        json={
            "username": settings.SUPERSET_USERNAME,
            "password": settings.SUPERSET_PASSWORD,
            "provider": "db"
        }
    )

    if auth_response.status_code != 200:
        raise HTTPException(status_code=500, detail="Failed to authenticate with Su

    access_token = auth_response.json()["access_token"]

    # Get guest token
    guest_token_response = requests.post(
        f"{settings.SUPERSET_URL}/api/v1/security/guest_token/",
        headers={"Authorization": f"Bearer {access_token}"},
        json={
            "resources": [{"type": "dashboard", "id": dashboard_id}],
            "user": {"username": current_user.username},
            "rls": []
        }
    )

    if guest_token_response.status_code != 200:
        raise HTTPException(status_code=500, detail="Failed to generate guest token

    return {"guest_token": guest_token_response.json()["token"]}
```

Step 4: Implement CSV Import Functionality

1. **CSV Import Endpoint**

```
python
```

```python
from fastapi import APIRouter, UploadFile, File, Depends, HTTPException
import pandas as pd
from sqlalchemy.orm import Session
from app.api.deps import get_db, get_current_user
from app.models.student import Student

router = APIRouter()

@router.post("/import-students/")
async def import_students(
    file: UploadFile = File(...),
    db: Session = Depends(get_db),
    current_user = Depends(get_current_user)
):
    # Check if user has permission
    if current_user.role != "admin":
        raise HTTPException(status_code=403, detail="Not authorized")

    # Process CSV file
    try:
        df = pd.read_csv(file.file)

        # Validate CSV structure
        required_columns = ["roll_number", "full_name", "batch", "admission_year"]
        if not all(column in df.columns for column in required_columns):
            raise HTTPException(status_code=400, detail="CSV missing required colum

        # Process each row
        students_added = 0
        errors = []

        for index, row in df.iterrows():
            try:
                # Check if student already exists
                existing = db.query(Student).filter(Student.roll_number == row["rol
                if existing:
                    errors.append(f"Student with roll number {row['roll_number']} a
                    continue

                # Create new student
                student = Student(
                    roll_number=row["roll_number"],
                    full_name=row["full_name"],
                    batch=row["batch"],
                    admission_year=int(row["admission_year"]),
                    email=row.get("email"),
```

```
                    phone=row.get("phone"),
                    current_semester=row.get("current_semester"),
                    cgpa=row.get("cgpa")
                )

                db.add(student)
                students_added += 1

            except Exception as e:
                errors.append(f"Error processing row {index}: {str(e)}")

        db.commit()

        return {
            "success": True,
            "students_added": students_added,
            "errors": errors
        }

    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error processing CSV: {str(e)
```

Step 5: Implement Google Forms Integration

1. **Create a Google Form with appropriate fields**
2. **Set up Google Form submission webhook**
3. **Create endpoint to receive form submissions**

python

```python
@router.post("/google-form-webhook/")
async def google_form_webhook(
    form_data: dict,
    db: Session = Depends(get_db)
):
    try:
        # Extract data from form submission
        student_id = form_data.get("student_id")
        title = form_data.get("achievement_title")
        description = form_data.get("achievement_description")
        achievement_type = form_data.get("achievement_type")
        achievement_date = form_data.get("achievement_date")

        # Validate student exists
        student = db.query(Student).filter(Student.id == student_id).first()
        if not student:
            return {"success": False, "error": "Student not found"}

        # Create achievement
        achievement = Achievement(
            student_id=student_id,
            title=title,
            description=description,
            achievement_type=achievement_type,
            achievement_date=achievement_date,
            is_verified=False  # Requires manual verification
        )

        db.add(achievement)
        db.commit()

        return {"success": True}

    except Exception as e:
        return {"success": False, "error": str(e)}
```

Frontend Development (Weeks 5-7)

Step 1: Set up Authentication and Routing

1. **Create Authentication Context**
   - Implement JWT storage and validation
   - Create protected routes

2. **Set up React Router**
    - Define all routes for your application
    - Implement navigation components

Step 2: Implement API Services

Create service modules for all API interactions:

1. **Authentication Service**
2. **Student Service**
3. **Achievement Service**
4. **Dashboard Service**
5. **Settings Service**

Step 3: Build UI Components

1. **Homepage**
    - Recent achievements section
    - Statistics dashboard
    - Recent students added
2. **Manage Students Page**
    - Student listing with search and filters
    - CSV import functionality
    - Student detail view
3. **Achievements Page**
    - Achievement listing with filters
    - Achievement detail view
    - Certificate display
4. **Dashboard Pages**
    - Superset dashboard embedding
    - Filtering and customization options
5. **Settings Page**
    - Website configuration options
    - User management (for admins)

Step 4: Implement Superset Dashboard Embedding

javascript

```jsx
// components/SupersetDashboard.js
import React, { useEffect, useState } from 'react';
import { embedDashboard } from '@superset-ui/embedded-sdk';
import { getGuestToken } from '../services/dashboardService';

const SupersetDashboard = ({ dashboardId }) => {
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const loadDashboard = async () => {
      try {
        setLoading(true);

        // Get guest token from your backend
        const { guest_token } = await getGuestToken(dashboardId);

        // Embed the dashboard
        const dashboard = await embedDashboard({
          id: dashboardId,
          supersetDomain: process.env.REACT_APP_SUPERSET_DOMAIN,
          mountPoint: document.getElementById('dashboard-container'),
          fetchGuestToken: () => guest_token,
          dashboardUiConfig: {
            hideTitle: false,
            hideChartControls: true,
            hideTab: false,
          }
        });

        setLoading(false);
      } catch (err) {
        console.error('Error loading dashboard:', err);
        setError('Failed to load dashboard');
        setLoading(false);
      }
    };

    loadDashboard();
  }, [dashboardId]);

  if (loading) return <div>Loading dashboard...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <div id="dashboard-container" style={{ width: '100%', height: '600px' }} />
```

```
  );
};

export default SupersetDashboard;
```

Step 5: Implement CSV Upload Component

javascript

```jsx
// components/CsvUpload.js
import React, { useState } from 'react';
import { Button, Typography, Paper, Box, CircularProgress, Alert } from '@mui/material
import CloudUploadIcon from '@mui/icons-material/CloudUpload';
import { importStudentsCsv } from '../services/studentService';

const CsvUpload = () => {
  const [file, setFile] = useState(null);
  const [loading, setLoading] = useState(false);
  const [result, setResult] = useState(null);
  const [error, setError] = useState(null);

  const handleFileChange = (e) => {
    if (e.target.files) {
      setFile(e.target.files[0]);
    }
  };

  const handleUpload = async () => {
    if (!file) {
      setError('Please select a file');
      return;
    }

    try {
      setLoading(true);
      setError(null);

      const formData = new FormData();
      formData.append('file', file);

      const response = await importStudentsCsv(formData);
      setResult(response);
      setFile(null);
    } catch (err) {
      setError(err.response?.data?.detail || 'Failed to upload file');
    } finally {
      setLoading(false);
    }
  };

  return (
    <Paper elevation={3} sx={{ p: 3, mb: 3 }}>
      <Typography variant="h6" gutterBottom>Import Students from CSV</Typography>

      <Typography variant="body2" color="textSecondary" paragraph>
```

```
  Upload a CSV file with the following columns: roll_number, full_name, batch, a
</Typography>

<Box sx={{ display: 'flex', alignItems: 'center', mb: 2 }}>
  <Button
    variant="outlined"
    component="label"
    startIcon={<CloudUploadIcon />}
  >
    Select CSV File
    <input
      type="file"
      accept=".csv"
      hidden
      onChange={handleFileChange}
    />
  </Button>
  {file && (
    <Typography variant="body2" sx={{ ml: 2 }}>
      {file.name}
    </Typography>
  )}
</Box>

<Button
  variant="contained"
  color="primary"
  onClick={handleUpload}
  disabled={!file || loading}
  sx={{ mr: 2 }}
>
  {loading ? <CircularProgress size={24} /> : 'Upload'}
</Button>

{error && (
  <Alert severity="error" sx={{ mt: 2 }}>
    {error}
  </Alert>
)}

{result && (
  <Alert severity={result.errors.length > 0 ? 'warning' : 'success'} sx={{ mt: 2
    Successfully added {result.students_added} students.
    {result.errors.length > 0 && (
      <>
        <Typography variant="body2" sx={{ mt: 1 }}>
          Errors encountered:
```

```
              </Typography>
              <ul>
                {result.errors.map((err, index) => (
                  <li key={index}>{err}</li>
                ))}
              </ul>
            </>
          )}
        </Alert>
      )}
    </Paper>
  );
};

export default CsvUpload;
```

Integration and Testing (Week 8)

Step 1: Connect Frontend and Backend

1. **Set up proxy in React**

   ```
   json

   // package.json
   {
     "proxy": "http://localhost:8000"
   }
   ```

2. **Configure CORS in FastAPI**

   ```
   python

   # main.py
   from fastapi.middleware.cors import CORSMiddleware

   app.add_middleware(
       CORSMiddleware,
       allow_origins=["http://localhost:3000"],
       allow_credentials=True,
       allow_methods=["*"],
       allow_headers=["*"],
   )
   ```

Step 2: Testing

1. **Backend Testing**
   - Unit tests for API endpoints

- Database integration tests

2. **Frontend Testing**
   - Component testing with React Testing Library
   - Integration testing with mock APIs
3. **End-to-End Testing**
   - Test complete user flows
   - Verify Superset integration

Deployment (Week 9)

Step 1: Prepare for Production

1. **Backend Preparation**
   - Configure environment variables
   - Set up logging
   - Optimize database queries
2. **Frontend Preparation**
   - Build production bundle

```bash
cd frontend
npm run build
```

Step 2: Deploy Application

1. **Set up production database**
2. **Deploy FastAPI backend**
   - Using a service like Heroku, AWS, or DigitalOcean
   - Configure with Gunicorn and Nginx
3. **Deploy React frontend**
   - Using services like Netlify, Vercel, or AWS S3
4. **Set up Apache Superset on production server**

Maintenance and Enhancements (Ongoing)

1. **Regular backups**
2. **Security updates**
3. **Performance monitoring**
4. **Feature enhancements based on user feedback**

Best Practices for Beginners

1. **Start small and iterate**
   - Begin with core functionality and add features incrementally
   - Use frequent git commits to track changes
2. **Use proper error handling**
   - Implement try/catch in JavaScript
   - Handle exceptions in Python
3. **Document your code**
   - Add comments to explain complex logic
   - Create API documentation
4. **Follow security best practices**
   - Store sensitive information in environment variables
   - Implement proper authentication and authorization
   - Validate all user inputs
5. **Test frequently**
   - Test each feature as you build it
   - Implement automated tests where possible