Project 4: User Authentication System - Key Concepts

1. Full-Stack Architecture with Backend-Frontend Separation

Concept Overview

Full-stack architecture with backend-frontend separation involves building applications with distinct backend (server) and frontend (client) components that communicate through well-defined APIs, enabling independent development, deployment, and scaling.

Problem It Solves

Monolithic applications where frontend and backend are tightly coupled create challenges for development teams, make testing difficult, limit technology choices, and create scaling bottlenecks.

Solution Approach

- API-First Design: Defining clear API contracts between frontend and backend
- Separation of Concerns: Backend focuses on data processing and business logic; frontend handles UI and user interaction
- Independent Development: Frontend and backend teams can work independently with minimal conflicts
- Technology Flexibility: Different technologies can be used for frontend and backend based on requirements

Implementation Insight

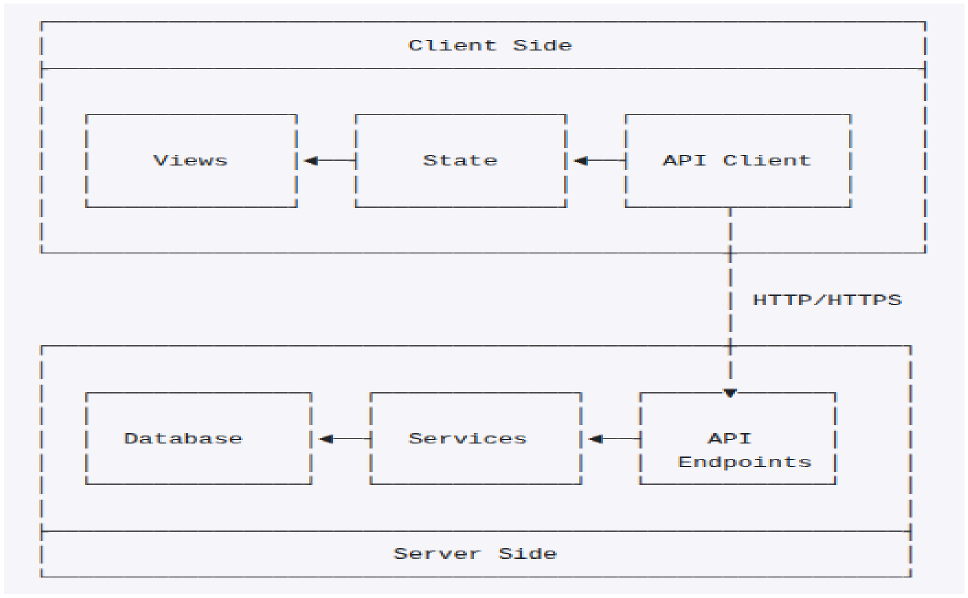
```
None
# Backend Structure (FastAPI)
backend/
├── app/
│   ├── __init__.py
│   ├── main.py              # Main FastAPI application
│   ├── api/
│   │   ├── __init__.py
│   │   ├── dependencies.py     # Shared API dependencies
│   │   └── routes/
│   │       ├── __init__.py
│   │       ├── auth.py        # Authentication endpoints
│   │       └── users.py       # User management endpoints
│   ├── core/
│   │   ├── __init__.py
│   │   ├── config.py          # Application configuration
│   │   └── security.py        # Security utilities
│   ├── db/
│   │   ├── __init__.py
│   │   ├── base.py            # Base DB setup
│   │   └── session.py         # DB session management
│   ├── models/
│   │   ├── __init__.py
│   │   └── user.py          # User DB model
│   └── schemas/
│       ├── __init__.py
│       ├── token.py         # Token schemas
│       └── user.py          # User data schemas
├── migrations/              # Alembic migrations
├── .env.example             # Example environment variables
├── requirements.txt         # Python dependencies
└── alembic.ini              # Migration configuration

# Frontend Structure (React + Vite)
frontend/
├── public/
│   └── favicon.ico          # Public assets
├── src/
│   ├── App.tsx              # Main application component
│   ├── main.tsx             # Application entry point
│   ├── api/
│   │   ├── index.ts         # API client setup
│   │   └── auth.ts          # Authentication API calls
│   ├── components/
│   │   ├── common/          # Shared UI components
│   │   ├── auth/            # Authentication components
│   │   └── layout/          # Layout components
│   ├── context/
│   │   └── AuthContext.tsx  # Authentication context
│   ├── hooks/
│   │   └── useAuth.ts       # Authentication hook
│   ├── pages/
│   │   ├── Home.tsx         # Public home page
│   │   ├── Login.tsx        # Login page
│   │   ├── Register.tsx     # Registration page
│   │   └── Dashboard.tsx    # Protected dashboard page
│   ├── routes/
│   │   └── index.tsx          # Routing configuration
│   └── utils/
│       └── token.ts           # Token utilities
├── .env.example               # Environment variables
├── package.json               # Frontend dependencies
├── tailwind.config.js         # Tailwind CSS configuration
└── vite.config.js             # Vite configuration
```

Implementation Guide: See Milestone 1: Backend Foundation & Database Setup and Milestone 2: Frontend Development for practical implementation steps.

Common Pitfalls

- Inconsistent API Contracts: Not maintaining consistent API structures and response formats
- CORS Configuration Issues: Incorrect Cross-Origin Resource Sharing settings blocking legitimate requests
- Over-fetching: Retrieving more data than needed for frontend operations
- Under-fetching: Making multiple API calls when data could be combined in one request
- Tight Coupling: Creating frontend components that depend on specific backend implementation details
- Authentication Flow Fragmentation: Inconsistent handling of auth tokens between frontend and backend

Architecture Diagram



2. Database Design and ORM Integration

Concept Overview

Database design with ORM (Object-Relational Mapping) integration involves creating efficient database schemas and using programming language objects to interact with the database rather than writing raw SQL.

Problem It Solves

Raw SQL queries are error-prone, difficult to maintain, and create security vulnerabilities like SQL injection. Additionally, manually mapping between database rows and application objects is tedious and error-prone.

Solution Approach

- Schema Design: Creating properly normalized database tables with appropriate relationships
- ORM Models: Defining Python classes that map to database tables
- Migration Management: Using tools like Alembic to handle schema evolution
- Type Safety: Leveraging ORM features for type checking and validation

Implementation Insight

Python

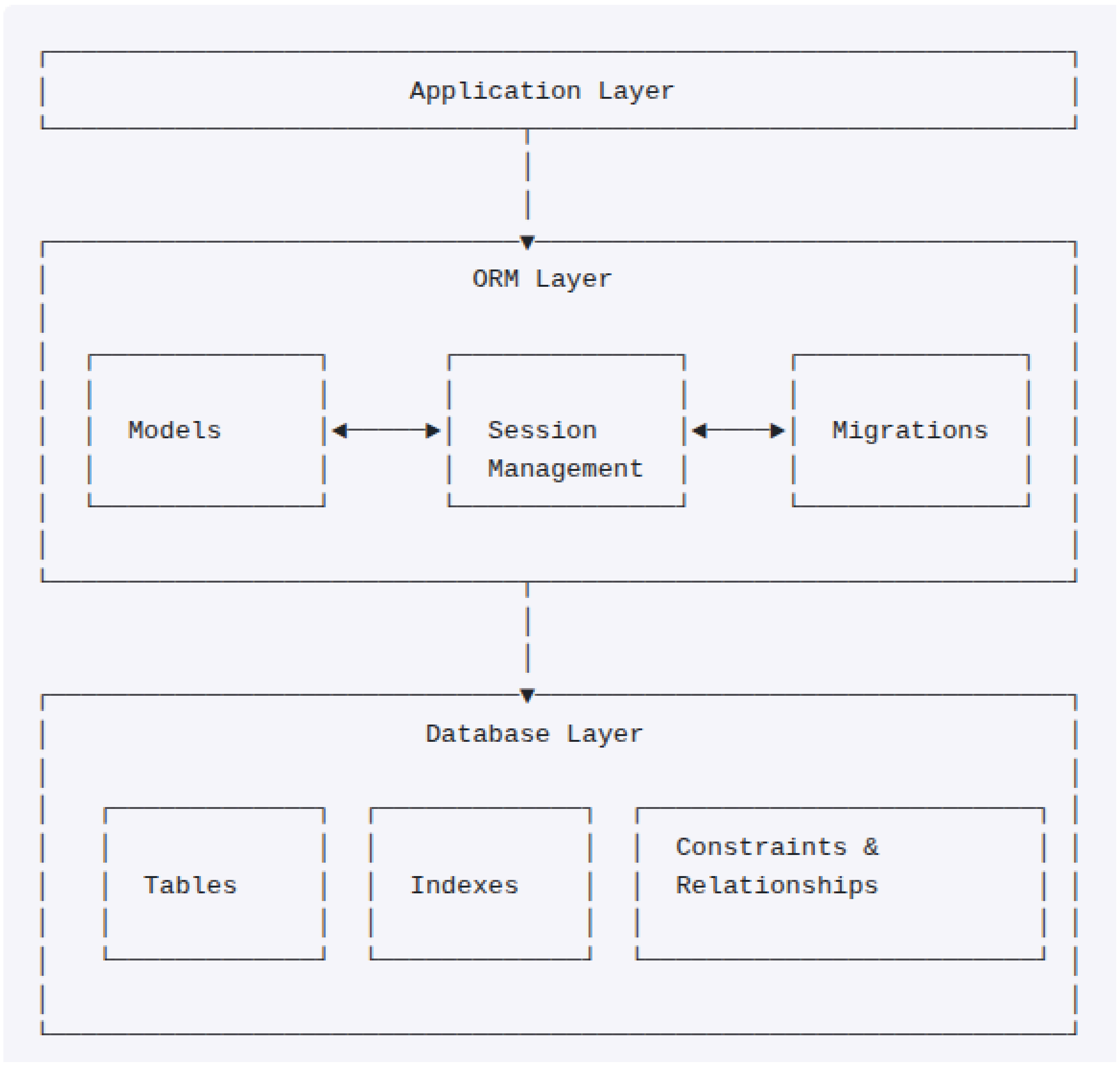
```python
# SQLAlchemy models
from sqlalchemy import Boolean, Column, Integer, String, DateTime
from sqlalchemy.sql import func
from app.db.base import Base
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True,
nullable=False)
    username = Column(String, unique=True, index=True,
nullable=False)
    hashed_password = Column(String, nullable=False)
    is_active = Column(Boolean, default=True)
    is_superuser = Column(Boolean, default=False)
    created_at = Column(DateTime(timezone=True),
server_default=func.now())
    updated_at = Column(DateTime(timezone=True),
onupdate=func.now())

    @classmethod
    def create(cls, db, *, email: str, username: str, password:
str):
        hashed_password = pwd_context.hash(password)
        user = cls(
            email=email,
            username=username,
            hashed_password=hashed_password
        )
        db.add(user)
        db.commit()
        db.refresh(user)
        return user

    def verify_password(self, plain_password: str) -> bool:
        return pwd_context.verify(plain_password,
self.hashed_password)

# Pydantic schemas for validation and serialization
from pydantic import BaseModel, EmailStr, Field, validator
from typing import Optional
import re

class UserBase(BaseModel):
    email: EmailStr
    username: str = Field(..., min_length=3, max_length=50)

    @validator("username")
    def username_alphanumeric(cls, v):
        if not re.match(r"^[a-zA-Z0-9_-]+$", v):
            raise ValueError("Username must be alphanumeric")
        return v
```

Common Pitfalls

- N+1 Query Problem: Loading related objects inefficiently with multiple database queries
- Lazy vs. Eager Loading Confusion: Not understanding when to use eager loading for related objects
- Session Management Issues: Not properly handling database sessions (opening/closing)
- Migration Conflicts: Improper handling of schema migrations causing data loss or corruption

- Object Detachment: Working with ORM objects outside their session context
- Field Type Mismatches: Inconsistencies between ORM model types and actual database column types

Architecture Diagram



3. JWT Authentication and Authorization

Concept Overview

JWT (JSON Web Token) authentication provides a stateless mechanism for verifying user identity and managing sessions across distributed systems without requiring server-side storage.

Problem It Solves

Traditional session-based authentication requires server memory to store session data, creating scaling challenges and making distributed systems complex. JWTs enable stateless authentication that works across multiple services.

Solution Approach

- Token Generation: Creating signed JWT tokens upon successful authentication
- Token Verification: Validating token signatures and expiration
- Claims Management: Including appropriate user data in token claims
- Token Refresh: Implementing token refresh mechanisms for longer sessions
- Authorization Middleware: Protecting routes based on token validation and user roles

Implementation Insight

Python
```python
# Backend JWT implementation
from datetime import datetime, timedelta
from typing import Optional
from jose import JWTError, jwt
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from pydantic import BaseModel

# Configuration
```

```
SECRET_KEY = "your-secret-key-stored-in-env"  # In production, use env var
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30
REFRESH_TOKEN_EXPIRE_DAYS = 7

# Token schemas
class Token(BaseModel):
    access_token: str
    refresh_token: str
    token_type: str

class TokenPayload(BaseModel):
    sub: Optional[int] = None  # User ID
    exp: Optional[int] = None  # Expiration time

# OAuth2 scheme for token extraction
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="auth/login")

def create_access_token(data: dict, expires_delta: Optional[timedelta] = None):
    """Create a new access token"""
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() +
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)

    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

def create_refresh_token(data: dict):
    """Create a new refresh token with longer expiry"""
    to_encode = data.copy()
    expire = datetime.utcnow() + timedelta(days=REFRESH_TOKEN_EXPIRE_DAYS)

    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

async def get_current_user(token: str = Depends(oauth2_scheme), db =
Depends(get_db)):
    """Dependency to get the current authenticated user from a token"""
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        # Decode the JWT
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        user_id: int = payload.get("sub")
        if user_id is None:
            raise credentials_exception

        # Create token payload model
        token_data = TokenPayload(sub=user_id, exp=payload.get("exp"))
    except JWTError:
        raise credentials_exception

    # Get user from database
    user = db.query(User).filter(User.id == token_data.sub).first()
    if user is None:
        raise credentials_exception
    # Check if token is expired
    if datetime.fromtimestamp(token_data.exp) < datetime.utcnow():
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Token expired",
            headers={"WWW-Authenticate": "Bearer"},
        )

    return user

# Login endpoint
@router.post("/login", response_model=Token)
async def login(form_data: OAuth2PasswordRequestForm = Depends(), db =
Depends(get_db)):
    """Authenticate user and return tokens"""
    # Find user by username
    user = db.query(User).filter(User.username == form_data.username).first()
    if not user or not user.verify_password(form_data.password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    # Create access and refresh tokens
    access_token = create_access_token(data={"sub": user.id})
    refresh_token = create_refresh_token(data={"sub": user.id})

    return {
        "access_token": access_token,
        "refresh_token": refresh_token,
        "token_type": "bearer"
    }
```

Implementation Guide: For practical JWT implementation, refer to JWT Authentication System in the Technical Implementation Guide.

4. Password Security and Hashing

Concept Overview

Password security and hashing involves storing passwords securely using cryptographic hashing algorithms designed specifically for passwords, rather than storing them in plain text or using insufficient encryption.

Problem It Solves

Storing plaintext passwords or using weak encryption creates significant security risks, as data breaches could expose user credentials, potentially affecting multiple services due to password reuse.

Solution Approach

- Modern Hashing Algorithms: Using bcrypt, Argon2, or similar algorithms designed for password hashing
- Salt Integration: Adding unique salts to prevent rainbow table attacks
- Work Factor Tuning: Adjusting computational cost to balance security and performance
- Secure Comparison: Using constant-time comparison to prevent timing attacks

Implementation Insight

Python
```python
from passlib.context import CryptContext

# Configure passlib with bcrypt
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
```

```
class PasswordManager:
    """Handles password hashing and verification"""

    @staticmethod
    def get_password_hash(password: str) -> str:
        """
        Hash a password using bcrypt

        Args:
            password: Plain-text password to hash

        Returns:
            Hashed password string
        """
        return pwd_context.hash(password)

    @staticmethod
    def verify_password(plain_password: str, hashed_password: str) -> bool:
        """
        Verify a password against a hash

        Args:
            plain_password: Plain-text password to check
            hashed_password: Previously hashed password

        Returns:
            True if password matches hash, False otherwise
        """
        return pwd_context.verify(plain_password, hashed_password)

    @staticmethod
    def is_password_strong(password: str) -> tuple[bool, str]:
```

```
    """
    Check password strength against security rules

    Args:
        password: Password to check

    Returns:
        Tuple of (is_valid, error_message)
    """
    if len(password) < 8:
        return False, "Password must be at least 8 characters long"

    if not any(char.isdigit() for char in password):
        return False, "Password must contain at least one digit"

    if not any(char.isupper() for char in password):
        return False, "Password must contain at least one uppercase letter"

    if not any(char.islower() for char in password):
        return False, "Password must contain at least one lowercase letter"

    if not any(char in "!@#$%^&*()-_=+[]{}|;:,.<>?/" for char in password):
        return False, "Password must contain at least one special character"

    return True, "Password is strong"

# Usage in API endpoint
@router.post("/register", response_model=UserResponse)
async def register(user_in: UserCreate, db = Depends(get_db)):
    """Register a new user"""
    # Check if email exists
    db_user = db.query(User).filter(User.email == user_in.email).first()
    if db_user:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Email already registered"
        )

    # Check if username exists
    db_user = db.query(User).filter(User.username == user_in.username).first()
    if db_user:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Username already taken"
        )

    # Validate password strength
    is_strong, error_message =
PasswordManager.is_password_strong(user_in.password)
    if not is_strong:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=error_message
        )

    # Create new user
    hashed_password = PasswordManager.get_password_hash(user_in.password)
    db_user = User(
        email=user_in.email,
        username=user_in.username,
        hashed_password=hashed_password
    )

    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user
```

5. React Authentication Context and Protected Routes

Concept Overview

React authentication context and protected routes provide a centralized state management system for user authentication status and restrict access to certain parts of an application based on that status.

Problem It Solves

Without a centralized authentication system, each component would need to implement authentication logic separately, leading to code duplication, inconsistencies, and potential security vulnerabilities.

Solution Approach

- Context API: Using React's Context to share authentication state across components
- Auth Provider: Creating a provider component that manages authentication logic
- Protected Route Components: Building route wrappers that redirect unauthenticated users
- Token Management: Handling token storage, validation, and refresh
- Login/Logout Functionality: Centralizing authentication operations

Implementation Insight

```None
// AuthContext.tsx
```

```
import React, { createContext, useContext, useState, useEffect } from 'react';
import { loginUser, refreshToken, getUserProfile } from '../api/auth';

// Define context types
interface AuthContextType {
  user: UserProfile | null;
  isAuthenticated: boolean;
  isLoading: boolean;
  login: (username: string, password: string) => Promise<void>;
  logout: () => void;
  error: string | null;
}

interface UserProfile {
  id: number;
  username: string;
  email: string;
}

// Create context with default values
const AuthContext = createContext<AuthContextType>({
  user: null,
  isAuthenticated: false,
  isLoading: true,
  login: async () => {},
  logout: () => {},
  error: null
});

// Custom hook to use auth context
export const useAuth = () => useContext(AuthContext);

// Auth Provider component
```

```
export const AuthProvider: React.FC<{ children: React.ReactNode }> = ({ children })
=> {
  const [user, setUser] = useState<UserProfile | null>(null);
  const [isLoading, setIsLoading] = useState<boolean>(true);
  const [error, setError] = useState<string | null>(null);

  // Check for existing token and validate on mount
  useEffect(() => {
    const checkAuth = async () => {
      const token = localStorage.getItem('access_token');
      if (!token) {
        setIsLoading(false);
        return;
      }

      try {
        // Try to get user profile with existing token
        const userProfile = await getUserProfile();
        setUser(userProfile);
      } catch (err) {
        // Try refreshing the token
        try {
          const refresh = localStorage.getItem('refresh_token');
          if (!refresh) {
            throw new Error('No refresh token available');
          }

          const newTokens = await refreshToken(refresh);
          localStorage.setItem('access_token', newTokens.access_token);
          localStorage.setItem('refresh_token', newTokens.refresh_token);

          // Get user profile with new token
          const userProfile = await getUserProfile();
          setUser(userProfile);
        } catch (refreshErr) {
          // If refresh fails, log out
          localStorage.removeItem('access_token');
          localStorage.removeItem('refresh_token');
          setUser(null);
        }
      } finally {
        setIsLoading(false);
      }
    };

    checkAuth();
  }, []);

  // Login function
  const login = async (username: string, password: string) => {
    setIsLoading(true);
    setError(null);

    try {
      const tokens = await loginUser(username, password);
      localStorage.setItem('access_token', tokens.access_token);
      localStorage.setItem('refresh_token', tokens.refresh_token);

      const userProfile = await getUserProfile();
      setUser(userProfile);
    } catch (err) {
      setError('Invalid username or password');
      setUser(null);
    } finally {
      setIsLoading(false);
    }
  };

  // Logout function
  const logout = () => {
    localStorage.removeItem('access_token');
    localStorage.removeItem('refresh_token');
    setUser(null);
  };

  const value = {
    user,
    isAuthenticated: !!user,
    isLoading,
    login,
    logout,
    error
  };

  return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>;
};

// Protected Route component
import { Navigate, Outlet, useLocation } from 'react-router-dom';

export const ProtectedRoute = () => {
  const { isAuthenticated, isLoading } = useAuth();
  const location = useLocation();

  if (isLoading) {
    return <div>Loading...</div>;
  }

  if (!isAuthenticated) {
    // Redirect to login page but save the current location
    return <Navigate to="/login" state={{ from: location }} replace />;
  }

  // Render child routes
  return <Outlet />;
};

// App routing
import { BrowserRouter, Routes, Route } from 'react-router-dom';

const App = () => {
  return (
    <AuthProvider>
      <BrowserRouter>
        <Routes>
          {/* Public routes */}
          <Route path="/" element={<HomePage />} />
          <Route path="/login" element={<LoginPage />} />
          <Route path="/register" element={<RegisterPage />} />

          {/* Protected routes */}
          <Route element={<ProtectedRoute />}>
            <Route path="/dashboard" element={<DashboardPage />} />
            <Route path="/profile" element={<ProfilePage />} />
            <Route path="/settings" element={<SettingsPage />} />
          </Route>

          {/* 404 route */}
          <Route path="*" element={<NotFoundPage />} />
        </Routes>
      </BrowserRouter>
    </AuthProvider>
  );
};
```

6. Modern Frontend Development with React, Vite, and Tailwind

Concept Overview

Modern frontend development combines React (component-based UI library), Vite (fast build tool), and Tailwind CSS (utility-first CSS framework) to create efficient, maintainable, and visually consistent web applications.

Problem It Solves

Traditional frontend development can be slow (long build times), inconsistent (varying design patterns), and difficult to maintain (complex CSS hierarchies and JavaScript bundles).

Solution Approach

- Component Architecture: Breaking UI into reusable, isolated components
- Fast Development: Using Vite for near-instantaneous HMR and builds
- Utility-First CSS: Leveraging Tailwind's atomic classes for consistent styling
- TypeScript Integration: Adding type safety to JavaScript for better developer experience
- Optimized Production: Generating highly optimized builds for production

Implementation Insight

```
None
// Login form component with Tailwind styling
import React, { useState } from 'react';
import { useAuth } from '../hooks/useAuth';
import { Link, useNavigate, useLocation } from 'react-router-dom';

export const LoginForm = () => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [isSubmitting, setIsSubmitting] = useState(false);
  const { login, error } = useAuth();
  const navigate = useNavigate();
  const location = useLocation();

  // Get the page to redirect to after login
  const from = (location.state as any)?.from?.pathname || '/dashboard';

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    if (isSubmitting) return;

    setIsSubmitting(true);
    try {
      await login(username, password);
      navigate(from, { replace: true });
    } finally {
      setIsSubmitting(false);
    }
  };

  return (
    <div className="min-h-screen flex items-center justify-center bg-gray-50 py-12
px-4 sm:px-6 lg:px-8">
      <div className="max-w-md w-full space-y-8">
        <div>
          <h2 className="mt-6 text-center text-3xl font-extrabold text-gray-900">
          Sign in to your account
        </h2>
        <p className="mt-2 text-center text-sm text-gray-600">
          Or{' '}
          <Link to="/register" className="font-medium text-indigo-600
hover:text-indigo-500">
            create a new account
          </Link>
        </p>
      </div>

      {error && (
        <div className="rounded-md bg-red-50 p-4">
          <div className="flex">
            <div className="flex-shrink-0">
              <svg className="h-5 w-5 text-red-400" viewBox="0 0 20 20"
fill="currentColor">
                <path fillRule="evenodd" d="M10 18a8 8 0 100-16 8 8 0 000 16zM8.707
7.293a1 1 0 00-1.414 1.414L8.586 10l-1.293 1.293a1 1 0 101.414 1.414L10
11.414l1.293 1.293a1 1 0 001.414-1.414L11.414 10l1.293-1.293a1 1 0
00-1.414-1.414L10 8.586 8.707 7.293z" clipRule="evenodd" />
              </svg>
            </div>
            <div className="ml-3">
              <h3 className="text-sm font-medium text-red-800">{error}</h3>
            </div>
          </div>
        </div>
      )}

      <form className="mt-8 space-y-6" onSubmit={handleSubmit}>
        <div className="rounded-md shadow-sm -space-y-px">
          <div>
            <label htmlFor="username" className="sr-only">Username</label>
            <input
              id="username"
              name="username"
              type="text"
              required
              value={username}
              onChange={(e) => setUsername(e.target.value)}
              className="appearance-none rounded-none relative block w-full px-3 py-2
border border-gray-300 placeholder-gray-500 text-gray-900 rounded-t-md
focus:outline-none focus:ring-indigo-500 focus:border-indigo-500 focus:z-10
sm:text-sm"
              placeholder="Username"
            />
          </div>
          <div>
            <label htmlFor="password" className="sr-only">Password</label>
            <input
              id="password"
              name="password"
              type="password"
              required
              value={password}
              onChange={(e) => setPassword(e.target.value)}
              className="appearance-none rounded-none relative block w-full px-3 py-2
border border-gray-300 placeholder-gray-500 text-gray-900 rounded-b-md
focus:outline-none focus:ring-indigo-500 focus:border-indigo-500 focus:z-10
sm:text-sm"
              placeholder="Password"
            />
          </div>
        </div>
```

```
          <div className="flex items-center justify-between">
            <div className="flex items-center">
              <input
                id="remember-me"
                name="remember-me"
                type="checkbox"
                className="h-4 w-4 text-indigo-600 focus:ring-indigo-500 border-gray-300
rounded"
              />
              <label htmlFor="remember-me" className="ml-2 block text-sm
text-gray-900">
                Remember me
              </label>
            </div>

            <div className="text-sm">
              <a href="#" className="font-medium text-indigo-600
hover:text-indigo-500">
                Forgot your password?
              </a>
            </div>
          </div>

          <div>
            <button
              type="submit"
              disabled={isSubmitting}
              className={`group relative w-full flex justify-center py-2 px-4 border
border-transparent text-sm font-medium rounded-md text-white bg-indigo-600
hover:bg-indigo-700 focus:outline-none focus:ring-2 focus:ring-offset-2
focus:ring-indigo-500 ${
                isSubmitting ? 'opacity-70 cursor-not-allowed' : ''
              }`}
          >
            {isSubmitting ? 'Signing in...' : 'Sign in'}
          </button>
        </div>
      </form>
    </div>
  </div>
 );
};
```

7. OAuth 2.0 and Social Authentication

Concept Overview

OAuth 2.0 is an industry-standard protocol for authorization that allows third-party applications to access user data from service providers (like Google, Facebook, or LinkedIn) without exposing user credentials.

Problem It Solves

Traditional authentication requires users to create and remember credentials for each service they use. Social login simplifies user onboarding by leveraging existing accounts, improving conversion rates and reducing friction.

Solution Approach

- OAuth 2.0 Flow: Implementing the authorization code flow with PKCE
- Provider Integration: Setting up application credentials with major providers
- User Identity Federation: Mapping social identities to application users
- Profile Synchronization: Maintaining consistent user data across providers

Implementation Insight

Python
```python
# Backend OAuth implementation with FastAPI
from fastapi import APIRouter, Depends, Request, Response, HTTPException, status
from fastapi.responses import RedirectResponse
from authlib.integrations.starlette_client import OAuth, OAuthError
from starlette.config import Config
from sqlalchemy.orm import Session
from app.db.session import get_db
from app.models.user import User, SocialAccount
from app.core.security import create_access_token, create_refresh_token
import uuid
import json

# OAuth configuration
config = Config('.env')
oauth = OAuth(config)

# Register OAuth providers
oauth.register(
    name='google',
    client_id=config('GOOGLE_CLIENT_ID', default=''),
    client_secret=config('GOOGLE_CLIENT_SECRET', default=''),

server_metadata_url='https://accounts.google.com/.well-known/openid-configuratio
n',
    client_kwargs={'scope': 'openid email profile'}
)

oauth.register(
    name='facebook',
    client_id=config('FACEBOOK_CLIENT_ID', default=''),
    client_secret=config('FACEBOOK_CLIENT_SECRET', default=''),
    access_token_url='https://graph.facebook.com/oauth/access_token',
    authorize_url='https://www.facebook.com/dialog/oauth',
    client_kwargs={'scope': 'email'},
    api_base_url='https://graph.facebook.com/v13.0/'
)

# OAuth router
router = APIRouter()

@router.get('/login/{provider}')
async def login(provider: str, request: Request):
    # Generate and store a random state to prevent CSRF
    state = str(uuid.uuid4())
    request.session['oauth_state'] = state

    # Redirect to the provider's authorization URL
    redirect_uri = request.url_for('auth:callback', provider=provider)
    return await oauth.create_client(provider).authorize_redirect(
        request, redirect_uri, state=state
    )

@router.get('/callback/{provider}')
async def callback(
    provider: str,
    request: Request,
    db: Session = Depends(get_db)
):
    # Verify state parameter to prevent CSRF
    stored_state = request.session.get('oauth_state')
    state_param = request.query_params.get('state')
    if not stored_state or stored_state != state_param:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Invalid state parameter"
        )
```

```
try:
    # Complete the OAuth flow by exchanging the auth code for tokens
    client = oauth.create_client(provider)
    token = await client.authorize_access_token(request)

    # Get user info from the provider
    userinfo = await client.userinfo(token=token)

    # Extract common profile fields (providers return different structures)
    if provider == 'google':
        provider_user_id = userinfo['sub']
        email = userinfo['email']
        name = userinfo.get('name')
    elif provider == 'facebook':
        provider_user_id = userinfo['id']
        email = userinfo.get('email')
        name = userinfo.get('name')
    else:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=f"Unsupported provider: {provider}"
        )

    # Check if this social account exists
    social_account = db.query(SocialAccount).filter(
        SocialAccount.provider == provider,
        SocialAccount.provider_user_id == provider_user_id
    ).first()

    if social_account:
        # Social account exists, get the user
        user = social_account.user
```

```
else:
    # Check if a user with this email exists
    user = db.query(User).filter(User.email == email).first()

    if user:
        # Link social account to existing user
        social_account = SocialAccount(
            user_id=user.id,
            provider=provider,
            provider_user_id=provider_user_id,
            provider_email=email,
            access_token=token.get('access_token'),
            refresh_token=token.get('refresh_token'),
            expires_at=token.get('expires_at')
        )
        db.add(social_account)
        db.commit()
    else:
        # Create a new user and social account
        user = User(
            email=email,
            username=email.split('@')[0],  # Simple username from email
            is_active=True,
            # Generate a secure random password that the user won't need
            hashed_password=uuid.uuid4().hex
        )
        db.add(user)
        db.commit()
        db.refresh(user)

        # Now create the social account
        social_account = SocialAccount(
            user_id=user.id,
                provider=provider,
                provider_user_id=provider_user_id,
                provider_email=email,
                access_token=token.get('access_token'),
                refresh_token=token.get('refresh_token'),
                expires_at=token.get('expires_at')
            )
            db.add(social_account)
            db.commit()

        # Create JWT tokens for our application auth
        access_token = create_access_token(data={"sub": str(user.id)})
        refresh_token = create_refresh_token(data={"sub": str(user.id)})

        # Redirect to frontend with tokens
        response =
RedirectResponse(url=f"/auth/success?access_token={access_token}&refresh_token
={refresh_token}")
        return response

    except OAuthError as error:
        return RedirectResponse(url=f"/auth/error?error={error.error}")
```

Frontend Integration

```
None
// Social login buttons component
import React from 'react';
import { useAuth } from '../hooks/useAuth';

interface SocialLoginProps {
  className?: string;
```

```
}

export const SocialLogin: React.FC<SocialLoginProps> = ({ className }) => {
  const { error } = useAuth();

  const handleSocialLogin = (provider: string) => {
    // Redirect to backend OAuth route
    window.location.href = `/api/auth/login/${provider}`;
  };

  return (
    <div className={`flex flex-col space-y-4 ${className}`}>
      <div className="flex items-center justify-center">
        <span className="px-2 text-gray-500">Or continue with</span>
      </div>

      <div className="flex space-x-4 justify-center">
        <button
          onClick={() => handleSocialLogin('google')}
          type="button"
          className="flex items-center justify-center px-4 py-2 border border-gray-300
rounded-md shadow-sm text-sm font-medium text-gray-700 bg-white
hover:bg-gray-50 focus:outline-none focus:ring-2 focus:ring-offset-2
focus:ring-indigo-500"
        >
          <svg className="h-5 w-5 mr-2" viewBox="0 0 24 24">
            {/* Google icon */}
            <path
              d="M12.545 10.239v3.821h5.445c-0.712 2.315-2.647 3.972-5.445 3.972-3.332
0-6.033-2.701-6.033-6.032s2.701-6.032 6.033-6.032c1.498 0 2.866 0.549 3.921
1.453l2.814-2.814c-1.787-1.676-4.139-2.701-6.735-2.701-5.522 0-10.001
4.478-10.001 10s4.478 10 10.001 10c8.396 0 10.249-7.85 9.426-11.748l-9.426 0.081z"
              fill="#EA4335"
            />
          </svg>
          Google
        </button>

        <button
          onClick={() => handleSocialLogin('facebook')}
          type="button"
          className="flex items-center justify-center px-4 py-2 border border-gray-300
rounded-md shadow-sm text-sm font-medium text-gray-700 bg-white
hover:bg-gray-50 focus:outline-none focus:ring-2 focus:ring-offset-2
focus:ring-indigo-500"
        >
          <svg className="h-5 w-5 mr-2" viewBox="0 0 24 24">
            {/* Facebook icon */}
            <path
              d="M24 12.073c0-6.627-5.373-12-12-12s-12 5.373-12 12c0 5.99 4.388 10.954
10.125 11.854v-8.385h-3.047v-3.47h3.047v-2.642c0-3.007 1.792-4.669 4.533-4.669
1.312 0 2.686.235 2.686.235v2.953h-1.514c-1.491 0-1.956.925-1.956
1.874v2.25h3.328l-.532 3.47h-2.796v8.385c5.738-.9 10.126-5.864 10.126-11.854z"
              fill="#1877F2"
            />
          </svg>
          Facebook
        </button>

        <button
          onClick={() => handleSocialLogin('linkedin')}
          type="button"
          className="flex items-center justify-center px-4 py-2 border border-gray-300
rounded-md shadow-sm text-sm font-medium text-gray-700 bg-white
hover:bg-gray-50 focus:outline-none focus:ring-2 focus:ring-offset-2
focus:ring-indigo-500"
```

```
       >
         <svg className="h-5 w-5 mr-2" viewBox="0 0 24 24">
           {/* LinkedIn icon */}
           <path
             d="M20.447 20.452h-3.554v-5.569c0-1.328-.027-3.037-1.852-3.037-1.853
0-2.136 1.445-2.136 2.939v5.667H9.351V9h3.414v1.561h.046c.477-.9 1.637-1.85
3.37-1.85 3.601 0 4.267 2.37 4.267 5.455v6.286zM5.337 7.433c-1.144
0-2.063-.926-2.063-2.065 0-1.138.92-2.063 2.063-2.063 1.14 0 2.064.925 2.064 2.063
0 1.139-.925 2.065-2.064 2.065zm1.782 13.019H3.555V9h3.564v11.452zM22.225
0H1.771C.792 0 0 .774 0 1.729v20.542C0 23.227.792 24 1.771 24h20.451C23.2 24 24
23.227 24 22.271V1.729C24 .774 23.2 0 22.222 0h.003z"
             fill="#0A66C2"
           />
         </svg>
         LinkedIn
       </button>
     </div>

     {error && (
       <div className="text-center text-red-500 text-sm mt-2">
         {error}
       </div>
     )}
   </div>
 );
};
```

Common Pitfalls

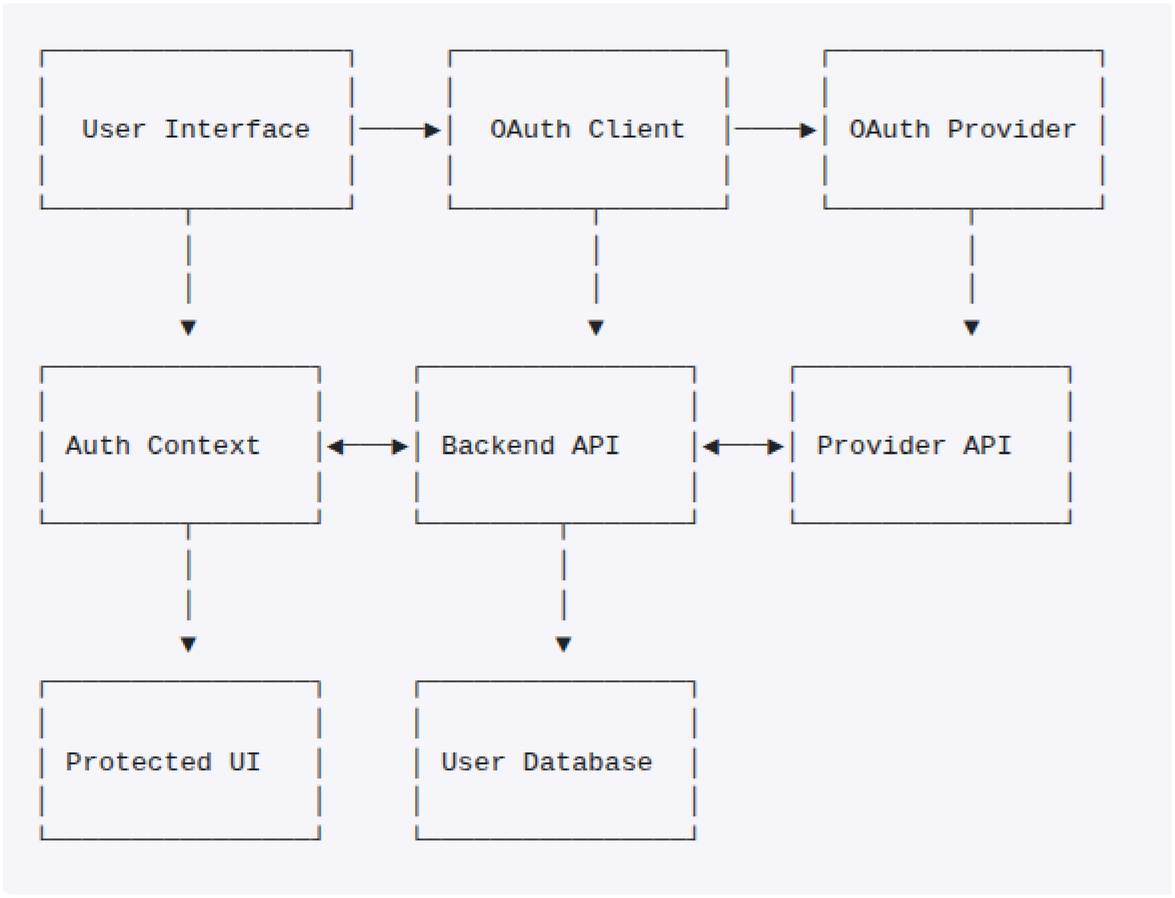
- Insufficient State Validation: Not properly validating the state parameter, risking CSRF attacks
- Scope Overreach: Requesting excessive permissions from providers

- User Data Inconsistency: Not handling profile data differences between providers
- Token Storage Security: Insecure storage of OAuth access/refresh tokens
- Account Linking Conflicts: Not properly handling cases where users have multiple social accounts
- Redirect URI Mismatches: Misconfigured redirect URIs between provider settings and application
- Session Fixation: Not regenerating session IDs during authentication flows

Cross-References

- Builds upon JWT Authentication and Authorization for token management
- Integrates with Database Design and ORM Integration for user storage
- Works with React Authentication Context for state management
- Enhances Password Security by providing alternative authentication methods

Architecture Diagram



Further Resources

- OAuth 2.0 Specification
- OpenID Connect
- OAuth 2.0 PKCE Flow
- Authlib Documentation
- Google OAuth Documentation
- Facebook Login Documentation
- LinkedIn OAuth Documentation