PsychSync - API Framework Setup Completion Guide

Epic: Core Infrastructure API Framework Setup

Assignee: Backend Developer 1 | **Effort:** 12h | **Priority:** High

Based on your existing project structure, here's what you need to complete the API Framework Setup task.

Current Structure Analysis

Your existing structure shows:

```
backend/
 ├── app/
|   ├── core/config.py ✓ (needs enhancement)
|   ├── db/ ✓ (good foundation)
|   ├── api/ ✓ (needs structure)
|   └── schemas/ ✓ (needs auth schemas)
 ├── main.py ✓ (needs FastAPI app setup)
 └── alembic/ ✓ (migrations ready)
```

Step 1: Complete Missing Authentication Infrastructure (3-4 hours)

1.1 Create Security Module

Create `backend/app/core/security.py`:

```python
```

```python
from datetime import datetime, timedelta
from typing import Any, Union, Optional
from jose import jwt, JWTError
from passlib.context import CryptContext
from fastapi import HTTPException, status
from app.core.config import settings

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

def create_access_token(
    subject: Union[str, Any], expires_delta: timedelta = None
) -> str:
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(
            minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES
        )
    to_encode = {"exp": expire, "sub": str(subject), "type": "access"}
    encoded_jwt = jwt.encode(to_encode, settings.SECRET_KEY, algorithm=settings.ALGORITHM)
    return encoded_jwt

def create_refresh_token(subject: Union[str, Any]) -> str:
    expire = datetime.utcnow() + timedelta(days=settings.REFRESH_TOKEN_EXPIRE_DAYS)
    to_encode = {"exp": expire, "sub": str(subject), "type": "refresh"}
    encoded_jwt = jwt.encode(to_encode, settings.SECRET_KEY, algorithm=settings.ALGORITHM)
    return encoded_jwt

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

def get_password_hash(password: str) -> str:
    return pwd_context.hash(password)

def verify_token(token: str, token_type: str = "access") -> Optional[str]:
    try:
        payload = jwt.decode(
            token, settings.SECRET_KEY, algorithms=[settings.ALGORITHM]
        )
        user_id: str = payload.get("sub")
        token_type_payload: str = payload.get("type")

        if user_id is None or token_type_payload != token_type:
            return None
        return user_id
    except JWTError:
        return None
```

1.2 Update Your Config.py

Enhance `backend/app/core/config.py`:

```python
from pydantic_settings import BaseSettings
from typing import Optional, List
import secrets
import os

class Settings(BaseSettings):
    # API Settings
    API_V1_STR: str = "/api/v1"
    PROJECT_NAME: str = "PsychSync API"
    PROJECT_DESCRIPTION: str = "Team Psychology & Optimization Platform API"
    VERSION: str = "1.0.0"

    # Security - NEW ADDITIONS
    SECRET_KEY: str = secrets.token_urlsafe(32)
    ACCESS_TOKEN_EXPIRE_MINUTES: int = 30
    REFRESH_TOKEN_EXPIRE_DAYS: int = 7
    ALGORITHM: str = "HS256"

    # Database - Use your existing setup
    DATABASE_URL: str = os.getenv("DATABASE_URL", "postgresql://user:password@localhost/psychsync")

    # CORS - Include your frontend port
    BACKEND_CORS_ORIGINS: List[str] = [
        "http://localhost:3000",
        "http://localhost:5173",  # Vite dev server
        "http://localhost:8000"
    ]

    # Environment
    ENVIRONMENT: str = "development"
    DEBUG: bool = True

    class Config:
        env_file = ".env"
        case_sensitive = True

settings = Settings()
```

1.3 Create Authentication Dependencies

Create `backend/app/api/dependencies/auth.py`:

python

```python
from typing import Optional
from fastapi import Depends, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from sqlalchemy.orm import Session
from app.db.session import get_db  # Use your existing db session
from app.core.security import verify_token
# Import your User model here - adjust path as needed
# from app.db.models.user import User

security = HTTPBearer()

async def get_current_user(
    credentials: HTTPAuthorizationCredentials = Depends(security),
    db: Session = Depends(get_db)
):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    token = credentials.credentials
    user_id = verify_token(token, "access")

    if user_id is None:
        raise credentials_exception

    # TODO: Query your User model once it's created
    # user = db.query(User).filter(User.id == user_id).first()
    # if user is None:
    #     raise credentials_exception

    # For now, return a mock user object
    return {"id": user_id, "is_active": True}

async def get_current_active_user(
    current_user = Depends(get_current_user),
):
    if not current_user.get("is_active"):
        raise HTTPException(status_code=400, detail="Inactive user")
    return current_user
```

Step 2: Create API Router Structure (2-3 hours)

2.1 Create V1 API Router

Create `backend/app/api/v1/__init__.py`:

```python
# Empty init file
```

Create `backend/app/api/v1/api.py`:

```python
from fastapi import APIRouter
from app.api.v1.endpoints import auth, users, health

api_router = APIRouter()

api_router.include_router(health.router, prefix="/health", tags=["health"])
api_router.include_router(auth.router, prefix="/auth", tags=["authentication"])
api_router.include_router(users.router, prefix="/users", tags=["users"])
```

2.2 Create Auth Schemas

Create `backend/app/schemas/auth.py`:

```python
from pydantic import BaseModel
from typing import Optional

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

class TokenPayload(BaseModel):
    sub: Optional[str] = None

class LoginRequest(BaseModel):
    email: str
    password: str

class RefreshTokenRequest(BaseModel):
    refresh_token: str
```

2.3 Create Auth Endpoints

Create `backend/app/api/v1/endpoints/auth.py`:

```python
```

```python
from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from app.db.session import get_db
from app.core.security import verify_password, create_access_token, create_refresh_token, verify_token
from app.schemas.auth import Token, LoginRequest, RefreshTokenRequest

router = APIRouter()

@router.post("/login", response_model=Token)
async def login(
    login_data: LoginRequest,
    db: Session = Depends(get_db)
):
    # TODO: Implement user authentication with your User model
    # For now, return mock tokens
    if login_data.email == "test@example.com" and login_data.password == "test123":
        access_token = create_access_token(subject="1")
        refresh_token = create_refresh_token(subject="1")

        return {
            "access_token": access_token,
            "refresh_token": refresh_token,
            "token_type": "bearer"
        }
    else:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect email or password",
        )

@router.post("/refresh", response_model=Token)
async def refresh_token(
    refresh_data: RefreshTokenRequest,
    db: Session = Depends(get_db)
):
    user_id = verify_token(refresh_data.refresh_token, "refresh")
    if not user_id:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid refresh token",
        )

    access_token = create_access_token(subject=user_id)
    refresh_token = create_refresh_token(subject=user_id)

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

2.4 Create Health and Users Endpoints

Create `backend/app/api/v1/endpoints/health.py`:

```python
from fastapi import APIRouter

router = APIRouter()

@router.get("/")
async def health_check():
    return {
        "status": "healthy",
        "service": "psychsync-api",
        "version": "1.0.0"
    }
```

Create `backend/app/api/v1/endpoints/users.py`:

```python
from fastapi import APIRouter, Depends
from app.api.dependencies.auth import get_current_active_user

router = APIRouter()

@router.get("/me")
async def read_user_me(
    current_user = Depends(get_current_active_user)
):
    return {
        "id": current_user["id"],
        "email": "user@example.com",  # TODO: Get from database
        "message": "This is a protected endpoint"
    }
```

Step 3: Update Your Main.py (1-2 hours)

Update `backend/main.py`:

python

```python
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from app.core.config import settings
from app.api.v1.api import api_router

# Create FastAPI app with enhanced configuration
app = FastAPI(
    title=settings.PROJECT_NAME,
    description=settings.PROJECT_DESCRIPTION,
    version=settings.VERSION,
    openapi_url=f"{settings.API_V1_STR}/openapi.json" if settings.DEBUG else None,
    docs_url=f"{settings.API_V1_STR}/docs" if settings.DEBUG else None,
    redoc_url=f"{settings.API_V1_STR}/redoc" if settings.DEBUG else None,
)

# Enhanced CORS middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=settings.BACKEND_CORS_ORIGINS,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Include API router
app.include_router(api_router, prefix=settings.API_V1_STR)

@app.get("/")
async def root():
    return {
        "message": "Welcome to PsychSync API",
        "version": settings.VERSION,
        "docs": f"{settings.API_V1_STR}/docs" if settings.DEBUG else "Not available in production"
    }

# For running with uvicorn
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Step 4: Enhanced OpenAPI Documentation (1-2 hours)

4.1 Add Custom OpenAPI Configuration

Update your `backend/main.py` to include custom OpenAPI:

```python
```

```
from fastapi.openapi.utils import get_openapi

def custom_openapi():
    if app.openapi_schema:
        return app.openapi_schema

    openapi_schema = get_openapi(
        title=settings.PROJECT_NAME,
        version=settings.VERSION,
        description="""
        ## PsychSync API

        The PsychSync API provides endpoints for team psychology analysis and optimization.

        ### Features
        * **Authentication**: JWT-based authentication system
        * **Team Management**: Create and manage teams and members
        * **Assessments**: Psychological assessment frameworks (MBTI, Big Five, DISC)
        * **Analytics**: Team compatibility and performance analytics
        * **Optimization**: AI-powered team optimization recommendations

        ### Authentication
        Most endpoints require authentication. To authenticate:
        1. Use `/api/v1/auth/login` to get access and refresh tokens
        2. Include the access token in the Authorization header as `Bearer <token>`
        3. Use `/api/v1/auth/refresh` to refresh expired tokens

        ### Getting Started
        1. Use the test credentials: `test@example.com` / `test123` to get tokens
        2. Use the "Authorize" button below to set your Bearer token
        3. Try the protected `/api/v1/users/me` endpoint
        """,
        routes=app.routes,
    )

    # Add security scheme
    openapi_schema["components"]["securitySchemes"] = {
        "HTTPBearer": {
            "type": "http",
            "scheme": "bearer",
            "bearerFormat": "JWT",
        }
    }

    app.openapi_schema = openapi_schema
    return app.openapi_schema
```

```
app.openapi = custom_openapi
```

Step 5: Install Required Dependencies (30 minutes)

From your `backend/` directory, install the required packages:

```bash
# Navigate to backend directory
cd backend

# Install authentication dependencies
pip install python-jose[cryptography] passlib[bcrypt] python-multipart

# Update requirements.txt
echo "python-jose[cryptography]==3.3.0" >> requirements.txt
echo "passlib[bcrypt]==1.7.4" >> requirements.txt
echo "python-multipart==0.0.6" >> requirements.txt
```

Step 6: Create Environment File (15 minutes)

Create `backend/.env`:

```env
SECRET_KEY=your-super-secret-key-change-this-in-production
DATABASE_URL=postgresql://your_existing_db_connection_string
ENVIRONMENT=development
DEBUG=true
```

Step 7: Test Your Implementation (1 hour)

7.1 Run the Server

From your `backend/` directory:

```bash
# Run the FastAPI server
python main.py

# Or use uvicorn directly
uvicorn main:app --reload --host 0.0.0.0 --port 8000
```

7.2 Test the API

1. Go to `http://localhost:8000/api/v1/docs`
2. Test the health endpoint: `GET /api/v1/health/`
3. Test authentication:
   - Use `POST /api/v1/auth/login` with email: `test@example.com`, password: `test123`
   - Copy the access_token from the response
   - Click "Authorize" button and paste: `Bearer your_access_token_here`
   - Test `GET /api/v1/users/me`

7.3 Expected Results

- Health endpoint returns: `{"status": "healthy", "service": "psychsync-api"}`
- Login returns access and refresh tokens
- Protected endpoint works with valid token

Next Steps After Completion

Once this is working:

1. **Connect to your database models** - Update the auth endpoints to use your actual User model
2. **Implement user registration** - Add signup endpoint
3. **Add more endpoints** - Based on your existing schemas and models
4. **Add comprehensive tests** - Create test files in your `tests/` directory
5. **Set up proper logging** - Add structured logging
6. **Configure production settings** - Separate dev/prod configs

Troubleshooting

Import errors: Make sure your Python path includes the `backend/` directory **Database connection**: Verify your DATABASE_URL in the .env file **CORS issues**: Check that your frontend URL is in BACKEND_CORS_ORIGINS **Token issues**: Ensure SECRET_KEY is set properly

Validation Checklist

- [ ] FastAPI server starts without errors
- [ ] OpenAPI docs accessible at `/api/v1/docs`
- [ ] Health endpoint responds correctly
- [ ] Login endpoint returns valid JWT tokens
- [ ] Protected endpoints require authentication
- [ ] CORS configured for frontend integration

- [ ] All imports resolve correctly

This setup provides the foundation for your authentication middleware and API documentation as specified in your project plan. The structure integrates with your existing codebase and can be extended with your specific business logic.