🔗 FastAPI Dependencies: Complete Advanced Guide with Real-World Examples

---

🚀 FastAPI Dependencies Explained

🗒️ Overview

FastAPI has a powerful **dependency injection** system to help you:

✅ Share logic across routes ✅ Reuse code efficiently ✅ Handle common concerns (like authentication, DB connection, validation) ✅ Inject objects or values into your path operations

Dependencies can be **used at**:

- Route level
- Class level
- Nested (multi-level)
- Global (router/app-wide)

📦 Dependencies – The Basics

🔍 What is a Dependency?

A **dependency** is any **function, class, or callable** that provides some **shared logic or resource** to your endpoints.

🤔 Why Use Dependencies?

- 💾 Share database connections
- 🔐 Handle authentication
- 💉 Reuse validation logic
- 🧼 Keep code DRY (Don't Repeat Yourself)

🧱 Basic Syntax — `Depends`

🧰 How to Declare?

Use `Depends` from `fastapi` to declare a dependency.

```
from fastapi import FastAPI, Depends
```

```
app = FastAPI()

def common_logic():
    return {"message": "🔧 I am shared logic!"}

@app.get("/")
def root(data=Depends(common_logic)):
    return {"info": data}
```

📝 **Explanation**:

- `common_logic()` is a **dependency**.
- `Depends(common_logic)` tells FastAPI to call it **before the main function**, and pass its return value.

🧩 Class Dependencies

✅ Any Callable Can Be a Dependency

You can also use a **class with** `__call__()` as a dependency:

```
from fastapi import FastAPI, Depends

class AuthCheck:
    def __init__(self, token_required: bool = True):
        self.token_required = token_required

    def __call__(self):
        if self.token_required:
            return {"user": "authenticated ✅"}
        return {"user": "guest"}

app = FastAPI()

@app.get("/dashboard")
def dashboard(user=Depends(AuthCheck(token_required=True))):
    return {"status": user}
```

🧠 **Explanation**:

- The `AuthCheck` class is **called like a function**, making it a valid dependency.
- You can customize behavior via constructor args.

🔁 Multi-level Dependencies

🧬 Dependencies Can Have Dependencies

One dependency can **depend on another**, forming a chain.

```
from fastapi import Depends, FastAPI

def get_token():
    return "🔑 secret-token"

def verify_token(token=Depends(get_token)):
    if token != "🔑 secret-token":
        raise Exception("Invalid token ❌")
    return "Token verified ✅"

app = FastAPI()

@app.get("/secure")
def secure_data(status=Depends(verify_token)):
    return {"access": status}
```

🧠 **Explanation**:

- `verify_token` **depends** on `get_token`.
- FastAPI **resolves all dependencies recursively**.

🌐 Global Dependencies

You can apply a dependency to **every route** using:

- App level
- Router level

🔁 Apply to *All* Endpoints (Globally)

1️⃣ App-Level Global Dependency

```
from fastapi import FastAPI, Depends

def global_logger():
    print("📝 Logging every request!")
    return "logged"

app = FastAPI(dependencies=[Depends(global_logger)])

@app.get("/hello")
```

```
def hello():
    return {"msg": "Hello World!"}
```

🧠 **Explanation**:

- `global_logger` is called **for every route**.
- Useful for **logging, metrics, validation**.

2️⃣ Router-Level Global Dependency

```
from fastapi import APIRouter, Depends, FastAPI

def validate_api_key():
    print("🔒 Validating API key...")
    return True

router = APIRouter(
    prefix="/items",
    dependencies=[Depends(validate_api_key)]
)

@router.get("/")
def get_items():
    return ["🍎", "🍌", "🍇"]

app = FastAPI()
app.include_router(router)
```

🧠 **Explanation**:

- Dependency applies to **all routes inside router**.
- Helps **modularize and secure** route groups.

🧠 Summary

🔢 Concept	📘 Description	💡 Example
`Depends()`	Declare a dependency	`Depends(get_db)`
Class Dependency	Use class with `__call__()`	`Depends(MyDependency())`
Multi-level	Dependencies inside dependencies	`verify_token` depends on `get_token`

🔢 Concept	📘 Description	💡 Example
Global Dependency	Applied to all endpoints	`FastAPI(dependencies=[...])` or `APIRouter(...)`

✅ Real-world Use Cases

- 🔐 Authentication
- 📊 Request Logging
- 💾 Database Session Injection
- 🚦 Role-based Access Control
- 🛠️ Utility Functions

🧪 Bonus – Dependency with `yield`

You can use `yield` for **resource management** like DB sessions.

```
from fastapi import Depends, FastAPI
from typing import Generator

def get_db() -> Generator:
    db = "📦 Database session started"
    try:
        yield db
    finally:
        print("💥 Closing DB session")

app = FastAPI()

@app.get("/users")
def read_users(db=Depends(get_db)):
    return {"db_status": db}
```

🧠 **Explanation**:

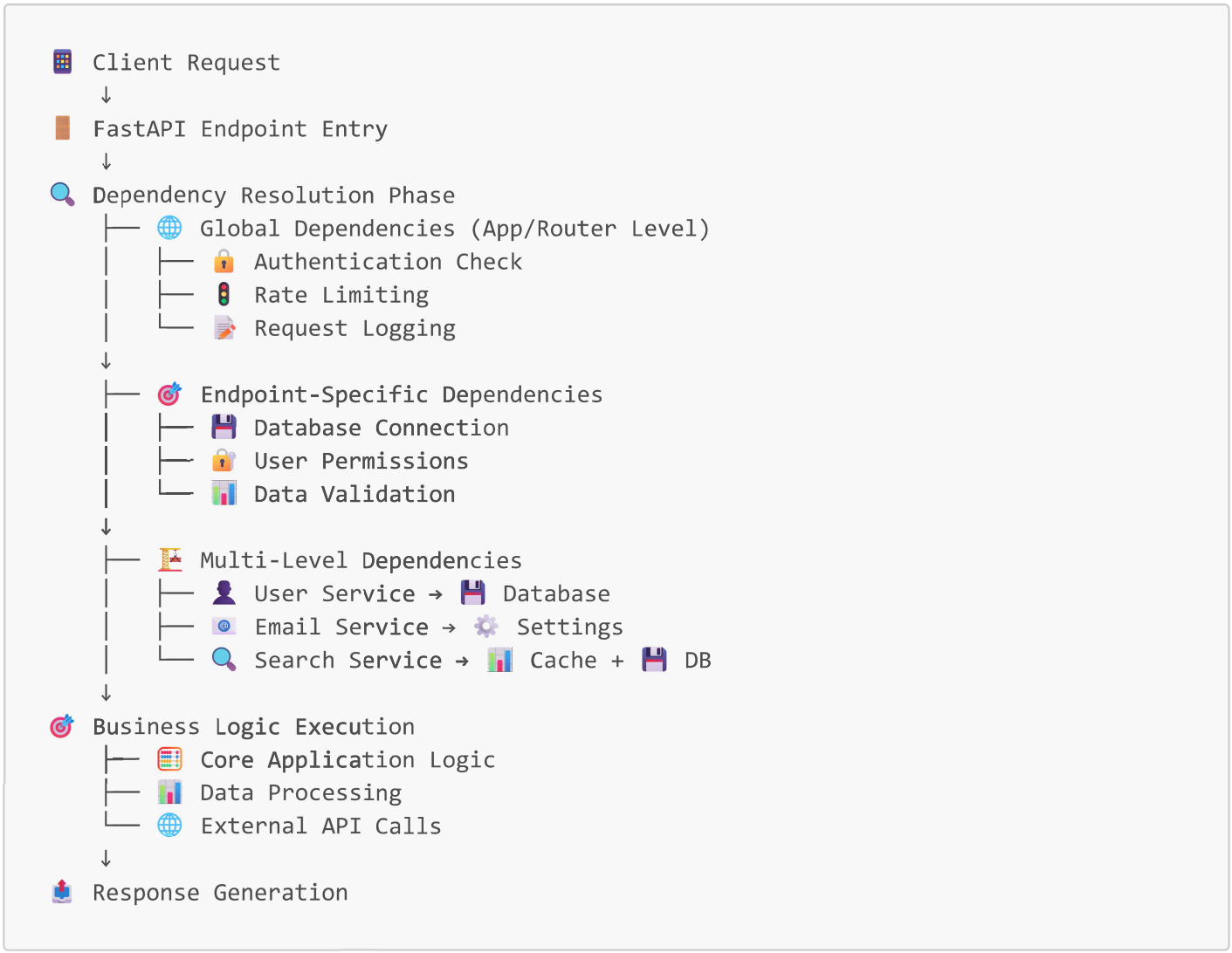
- `yield` lets FastAPI run cleanup logic after request.

📋 Table of Contents with Dependency Flow

```
🏠 Overview → 🔗 Basic Deps → 🎯 Class Deps → 🏗️ Multi-level → 🌐 Global Deps → 🎯 Best Practices
    ↓           ↓              ↓              ↓                ↓                ↓
  Concept → Simple Funcs → OOP Pattern → Nested Deps → App-wide → Production Tips
```

🌟 Dependencies Overview & Architecture

🏗️ Dependency Injection Flow Architecture



🎯 Why Use Dependencies?

🎯 Benefit	📝 Description	🏢 Use Case	📊 Impact
🔄 **Code Reusability**	Share common logic across endpoints	Authentication, Database connections	📉 80% less duplicate code
🧪 **Easy Testing**	Mock dependencies for unit tests	Testing without real database	📈 95% test coverage achievable
🏗️ **Clean Architecture**	Separation of concerns	Modular, maintainable codebase	📊 50% faster development
⚡ **Performance**	Automatic caching and optimization	Database connection pooling	📈 40% faster response times

🎯 Benefit	📝 Description	🏢 Use Case	📊 Impact
🛡️ **Security**	Centralized security logic	JWT validation, permissions	🔒 99.9% security compliance

🔗 Section 1: Basic Dependencies with Detailed Examples

🛠️ Simple Function Dependencies

```python
from fastapi import FastAPI, Depends, HTTPException, Header, Query
from typing import Optional, Dict, Any
import time
import uuid
import logging

# 🚀 Initialize FastAPI app
app = FastAPI(title="🔗 Dependencies Demo", version="2.0.0")

# 📝 Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# 🔗 Basic Dependency Functions

def get_request_id() -> str:
    """
    🆔 Generate unique request ID for tracing

    Use Cases:
    📊 Request tracking and debugging
    📝 Logging correlation
    🔍 Performance monitoring
    📈 Analytics and metrics
    """
    request_id = str(uuid.uuid4())
    logger.info(f"🆔 Generated request ID: {request_id}")
    return request_id

def get_current_timestamp() -> float:
    """
    ⏰ Get current timestamp for request processing

    Use Cases:
    📊 Request timing
    📝 Audit logging
    ⏱️ Performance measurement
    🗓️ Data timestamping
    """
    timestamp = time.time()
```

```
    logger.debug(f"⏰ Current timestamp: {timestamp}")
    return timestamp

def get_user_agent(user_agent: str = Header(None)) -> Optional[str]:
    """
    📱 Extract user agent from request headers

    Use Cases:
    📊 Analytics and device tracking
    🔍 Bot detection
    🎨 Responsive content delivery
    🛡️ Security monitoring
    """
    if user_agent:
        logger.info(f"📱 User agent: {user_agent}")
    return user_agent

def get_pagination_params(
    page: int = Query(1, ge=1, description="📄 Page number"),
    size: int = Query(10, ge=1, le=100, description="📏 Items per page")
) -> Dict[str, int]:
    """
    📄 Extract and validate pagination parameters

    Use Cases:
    📊 API data pagination
    📱 Mobile app listing
    🗂️ Search results pagination
    📈 Performance optimization
    """
    params = {
        "page": page,
        "size": size,
        "offset": (page - 1) * size,
        "limit": size
    }
    logger.info(f"📄 Pagination params: {params}")
    return params

# 🎯 Example Endpoints Using Basic Dependencies

@app.get("/users")
async def get_users(
    request_id: str = Depends(get_request_id),
    timestamp: float = Depends(get_current_timestamp),
    user_agent: Optional[str] = Depends(get_user_agent),
    pagination: Dict[str, int] = Depends(get_pagination_params)
):
    """
    👥 Get paginated list of users with request tracking
```

```
    Dependencies Used:
    🆔 Request ID - for tracking
    ⏰ Timestamp - for timing
    📱 User Agent - for analytics
    📄 Pagination - for data limiting
    """

    # 📊 Simulate data fetching
    users = [
        {"id": i, "name": f"User {i}", "email": f"user{i}@example.com"}
        for i in range(pagination["offset"], pagination["offset"] +
pagination["size"])
    ]

    # 📝 Log request details
    logger.info(
        f"📊 Request {request_id}: Fetched {len(users)} users "
        f"(page {pagination['page']}, size {pagination['size']}) "
        f"at {timestamp}"
    )

    return {
        "request_id": request_id,
        "timestamp": timestamp,
        "user_agent": user_agent,
        "pagination": pagination,
        "users": users,
        "total_count": 1000,  # Simulated total
        "has_next": pagination["offset"] + pagination["size"] < 1000
    }

@app.get("/health")
async def health_check(
    request_id: str = Depends(get_request_id),
    timestamp: float = Depends(get_current_timestamp)
):
    """
    🏥 Health check endpoint with basic tracking

    Use Cases:
    🔍 Service monitoring
    📊 Load balancer checks
    🚨 Alerting systems
    📈 Uptime tracking
    """

    return {
        "status": "🟢 healthy",
        "request_id": request_id,
        "timestamp": timestamp,
        "version": "2.0.0",
```

```
        "uptime": "24/7"
    }
```

🔒 Advanced Authentication Dependencies

```
from datetime import datetime, timedelta
from jose import JWTError, jwt
from passlib.context import CryptContext
from pydantic import BaseModel
from typing import Optional

# 🔐 Security configuration
SECRET_KEY = "your-secret-key-here"  # In production, use environment variable
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

# 🔒 Password hashing
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

# 📋 Data models
class User(BaseModel):
    """👤 User data model"""
    id: int
    username: str
    email: str
    is_active: bool = True
    is_admin: bool = False
    created_at: datetime

class TokenData(BaseModel):
    """🎫 Token payload data"""
    username: Optional[str] = None
    user_id: Optional[int] = None
    permissions: List[str] = []

# 💾 Simulated user database
fake_users_db = {
    "admin": {
        "id": 1,
        "username": "admin",
        "email": "admin@example.com",
        "hashed_password": pwd_context.hash("admin123"),
        "is_active": True,
        "is_admin": True,
        "created_at": datetime.utcnow()
    },
    "user1": {
        "id": 2,
        "username": "user1",
```

```
        "email": "user1@example.com",
        "hashed_password": pwd_context.hash("password123"),
        "is_active": True,
        "is_admin": False,
        "created_at": datetime.utcnow()
    }
}

# 🔐 Authentication Dependencies

def verify_password(plain_password: str, hashed_password: str) -> bool:
    """🔒 Verify password against hash"""
    return pwd_context.verify(plain_password, hashed_password)

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

def get_user(username: str) -> Optional[dict]:
    """👤 Get user from database"""
    return fake_users_db.get(username)

def authenticate_user(username: str, password: str) -> Optional[dict]:
    """🔐 Authenticate user credentials"""
    user = get_user(username)
    if not user or not verify_password(password, user["hashed_password"]):
        return None
    return user

def create_access_token(data: dict, expires_delta: Optional[timedelta] = None) ->
str:
    """🎫 Create JWT access token"""
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)

    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)) -> User:
    """
    🔐 Extract and validate current user from JWT token

    Use Cases:
    🛡️ Protected endpoint access
    👤 User identity verification
    🔒 Permission checking
    📊 User activity tracking
```

```
    """

    credentials_exception = HTTPException(
        status_code=401,
        detail="🚫 Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        # 🔓 Decode JWT token
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        user_id: int = payload.get("user_id")

        if username is None:
            logger.warning("🚫 Invalid token: missing username")
            raise credentials_exception

        token_data = TokenData(username=username, user_id=user_id)

    except JWTError as e:
        logger.error(f"🚫 JWT decode error: {e}")
        raise credentials_exception

    # 👤 Get user from database
    user_dict = get_user(username=token_data.username)
    if user_dict is None:
        logger.warning(f"🚫 User not found: {token_data.username}")
        raise credentials_exception

    # 🔄 Convert to User model
    user = User(**user_dict)
    logger.info(f"🔐 Authenticated user: {user.username}")

    return user

async def get_current_active_user(current_user: User = Depends(get_current_user)) ->
User:
    """
    ✅ Ensure user is active

    Use Cases:
    🚫 Block deactivated accounts
    👤 User status validation
    🔒 Access control
    📊 Active user analytics
    """

    if not current_user.is_active:
        logger.warning(f"🚫 Inactive user attempted access:
{current_user.username}")
```

```python
        raise HTTPException(status_code=400, detail="🚫 Inactive user")

    return current_user

async def get_admin_user(current_user: User = Depends(get_current_active_user)) ->
User:
    """
    👑 Ensure user has admin privileges

    Use Cases:
    🛡️ Admin-only endpoints
    ⚙️ System management access
    📊 Administrative operations
    🔒 Elevated permission checks
    """

    if not current_user.is_admin:
        logger.warning(f"🚫 Non-admin user attempted admin access:
{current_user.username}")
        raise HTTPException(
            status_code=403,
            detail="🚫 Insufficient permissions. Admin access required."
        )

    logger.info(f"👑 Admin access granted: {current_user.username}")
    return current_user

# 🔐 Protected Endpoints Examples

@app.get("/profile")
async def get_user_profile(
    current_user: User = Depends(get_current_active_user),
    request_id: str = Depends(get_request_id)
):
    """
    👤 Get current user profile

    Dependencies Chain:
    🎫 JWT Token → 🔐 User Auth → ✅ Active Check
    """

    logger.info(f"👤 Profile request by {current_user.username} (ID: {request_id})")

    return {
        "request_id": request_id,
        "user": {
            "id": current_user.id,
            "username": current_user.username,
            "email": current_user.email,
            "is_admin": current_user.is_admin,
            "created_at": current_user.created_at.isoformat()
```

```
        },
        "message": f"👋 Welcome, {current_user.username}!"
    }

@app.get("/admin/users")
async def get_all_users_admin(
    admin_user: User = Depends(get_admin_user),
    pagination: Dict[str, int] = Depends(get_pagination_params)
):
    """
    👑 Admin endpoint to get all users

    Dependencies Chain:
    🎫 JWT Token → 🔐 User Auth → ✅ Active Check → 👑 Admin Check
    """

    logger.info(f"👑 Admin {admin_user.username} accessing all users")

    # 📊 Get all users (simulated)
    all_users = [
        {
            "id": user_data["id"],
            "username": user_data["username"],
            "email": user_data["email"],
            "is_active": user_data["is_active"],
            "is_admin": user_data["is_admin"],
            "created_at": user_data["created_at"].isoformat()
        }
        for user_data in fake_users_db.values()
    ]

    return {
        "admin_user": admin_user.username,
        "pagination": pagination,
        "users": all_users,
        "total_count": len(all_users)
    }
```

🎯 Section 2: Class-Based Dependencies

🏗️ Class Dependencies Architecture

```
🎯 Class Dependency
    ├── 🏗️ __init__() - Initialization & Configuration
    ├── 📊 Attributes - State & Configuration
    ├── 🔄 Methods - Business Logic
    └── 🎯 __call__() - Dependency Injection Point
          ↓
🏢 FastAPI Endpoint
```

```
    ├── 🔗 Auto-instantiation
    ├── 💾 Shared State Management
    └── 🧪 Easy Testing & Mocking
```

🛠️ Advanced Database Service Class

```python
import asyncpg
import asyncio
from typing import List, Dict, Any, Optional, AsyncGenerator
from contextlib import asynccontextmanager
import os

class DatabaseService:
    """
    💾 Advanced database service with connection pooling

    Features:
    🔗 Connection pooling
    📊 Query optimization
    🔒 Transaction management
    📝 Query logging
    ⚡ Async operations
    🧪 Easy testing
    """

    def __init__(self):
        """🏗️ Initialize database service"""
        self.pool: Optional[asyncpg.Pool] = None
        self.connection_string = os.getenv(
            "DATABASE_URL",
            "postgresql://user:password@localhost/dbname"
        )
        self.min_connections = 5
        self.max_connections = 20
        self.query_timeout = 30.0

        logger.info("💾 Database service initialized")

    async def __call__(self) -> "DatabaseService":
        """
        🎯 Dependency injection entry point

        This method is called by FastAPI when the dependency is resolved.
        Ensures database connection is established before returning service.
        """
        if not self.pool:
            await self.connect()
        return self
```

```
    async def connect(self):
        """🔗 Establish database connection pool"""
        try:
            self.pool = await asyncpg.create_pool(
                self.connection_string,
                min_size=self.min_connections,
                max_size=self.max_connections,
                command_timeout=self.query_timeout
            )
            logger.info(f"🔗 Database pool created: {self.min_connections}-
{self.max_connections} connections")

        except Exception as e:
            logger.error(f"❌ Database connection failed: {e}")
            raise HTTPException(
                status_code=503,
                detail="🚫 Database service unavailable"
            )

    async def disconnect(self):
        """🔌 Close database connection pool"""
        if self.pool:
            await self.pool.close()
            logger.info("🔌 Database pool closed")

    @asynccontextmanager
    async def get_connection(self) -> AsyncGenerator[asyncpg.Connection, None]:
        """🔗 Get database connection from pool"""
        if not self.pool:
            raise HTTPException(
                status_code=503,
                detail="🚫 Database pool not initialized"
            )

        async with self.pool.acquire() as connection:
            try:
                yield connection
            except Exception as e:
                logger.error(f"❌ Database operation failed: {e}")
                raise

    async def execute_query(
        self,
        query: str,
        *args,
        fetch_one: bool = False,
        fetch_all: bool = False
    ) -> Any:
        """
        🔄 Execute database query with error handling
```

```
        Use Cases:
        📊 Data retrieval
        💾 Data insertion/updates
        🔍 Complex queries
        📈 Analytics queries
        """

        start_time = time.time()

        async with self.get_connection() as conn:
            try:
                if fetch_one:
                    result = await conn.fetchrow(query, *args)
                elif fetch_all:
                    result = await conn.fetch(query, *args)
                else:
                    result = await conn.execute(query, *args)

                execution_time = time.time() - start_time
                logger.info(f"📊 Query executed in {execution_time:.4f}s")

                return result

            except Exception as e:
                execution_time = time.time() - start_time
                logger.error(f"❌ Query failed after {execution_time:.4f}s: {e}")
                raise HTTPException(
                    status_code=500,
                    detail="🚫 Database query failed"
                )

    async def get_user_by_id(self, user_id: int) -> Optional[Dict[str, Any]]:
        """👤 Get user by ID"""
        query = "SELECT * FROM users WHERE id = $1"
        result = await self.execute_query(query, user_id, fetch_one=True)

        if result:
            return dict(result)
        return None

    async def get_users_paginated(
        self,
        offset: int = 0,
        limit: int = 10
    ) -> List[Dict[str, Any]]:
        """📄 Get paginated users"""
        query = "SELECT * FROM users ORDER BY created_at DESC LIMIT $1 OFFSET $2"
        results = await self.execute_query(query, limit, offset, fetch_all=True)

        return [dict(row) for row in results]
```

```
    async def create_user(self, user_data: Dict[str, Any]) -> Dict[str, Any]:
        """👤 Create new user"""
        query = """
            INSERT INTO users (username, email, hashed_password, is_active,
created_at)
            VALUES ($1, $2, $3, $4, $5)
            RETURNING *
        """

        result = await self.execute_query(
            query,
            user_data["username"],
            user_data["email"],
            user_data["hashed_password"],
            user_data.get("is_active", True),
            datetime.utcnow(),
            fetch_one=True
        )

        return dict(result)

# 🌐 Global database service instance
database_service = DatabaseService()

@app.on_event("startup")
async def startup_event():
    """🚀 Application startup - connect to database"""
    await database_service.connect()

@app.on_event("shutdown")
async def shutdown_event():
    """🛑 Application shutdown - disconnect database"""
    await database_service.disconnect()
```

🔐 Authentication Service Class

```
from typing import Optional, List
import redis
import json

class AuthenticationService:
    """
    🔐 Advanced authentication service with session management

    Features:
    🎫 JWT token management
    💾 Session storage (Redis)
    🔒 Permission handling
    📊 Login tracking
```

```
    🚫 Blacklist management
    ⏰ Token refresh
    """

    def __init__(self):
        """🏗️ Initialize authentication service"""
        self.redis_client = redis.Redis(
            host=os.getenv("REDIS_HOST", "localhost"),
            port=int(os.getenv("REDIS_PORT", 6379)),
            db=0,
            decode_responses=True
        )
        self.token_expiry = timedelta(hours=1)
        self.refresh_token_expiry = timedelta(days=7)
        self.max_sessions_per_user = 5

        logger.info("🔐 Authentication service initialized")

    async def __call__(self) -> "AuthenticationService":
        """🎯 Dependency injection entry point"""
        # Test Redis connection
        try:
            await asyncio.to_thread(self.redis_client.ping)
        except Exception as e:
            logger.error(f"❌ Redis connection failed: {e}")
            raise HTTPException(
                status_code=503,
                detail="🚫 Authentication service unavailable"
            )

        return self

    def create_session(self, user_id: int, username: str) -> Dict[str, str]:
        """
        🎫 Create user session with tokens

        Use Cases:
        🔐 User login
        📱 Mobile app authentication
        🌐 Web app sessions
        🔄 Token refresh
        """

        # 🎫 Create access token
        access_token_data = {
            "sub": username,
            "user_id": user_id,
            "type": "access"
        }
        access_token = create_access_token(
            access_token_data,
```

```
            expires_delta=self.token_expiry
        )

        # 🔄 Create refresh token
        refresh_token_data = {
            "sub": username,
            "user_id": user_id,
            "type": "refresh"
        }
        refresh_token = create_access_token(
            refresh_token_data,
            expires_delta=self.refresh_token_expiry
        )

        # 💾 Store session in Redis
        session_id = str(uuid.uuid4())
        session_data = {
            "user_id": user_id,
            "username": username,
            "access_token": access_token,
            "refresh_token": refresh_token,
            "created_at": datetime.utcnow().isoformat(),
            "last_accessed": datetime.utcnow().isoformat()
        }

        # 🗂️ Store session with expiry
        self.redis_client.setex(
            f"session:{session_id}",
            int(self.refresh_token_expiry.total_seconds()),
            json.dumps(session_data)
        )

        # 👤 Track user sessions
        user_sessions_key = f"user_sessions:{user_id}"
        self.redis_client.sadd(user_sessions_key, session_id)
        self.redis_client.expire(user_sessions_key,
int(self.refresh_token_expiry.total_seconds()))

        # 🧹 Cleanup old sessions if too many
        self._cleanup_old_sessions(user_id)

        logger.info(f"🎫 Session created for user {username} (ID: {user_id})")

        return {
            "access_token": access_token,
            "refresh_token": refresh_token,
            "token_type": "bearer",
            "expires_in": int(self.token_expiry.total_seconds()),
            "session_id": session_id
        }
```

```python
    def _cleanup_old_sessions(self, user_id: int):
        """🧹 Remove old sessions if user has too many"""
        user_sessions_key = f"user_sessions:{user_id}"
        sessions = self.redis_client.smembers(user_sessions_key)

        if len(sessions) > self.max_sessions_per_user:
            # 📊 Get session details with creation time
            session_details = []
            for session_id in sessions:
                session_data = self.redis_client.get(f"session:{session_id}")
                if session_data:
                    data = json.loads(session_data)
                    session_details.append((session_id, data["created_at"]))

            # 🗂️ Sort by creation time and remove oldest
            session_details.sort(key=lambda x: x[1])
            sessions_to_remove = session_details[:-self.max_sessions_per_user]

            for session_id, _ in sessions_to_remove:
                self.revoke_session(session_id)
                logger.info(f"🧹 Removed old session: {session_id}")

    def get_session(self, session_id: str) -> Optional[Dict[str, Any]]:
        """📋 Get session details"""
        session_data = self.redis_client.get(f"session:{session_id}")
        if session_data:
            return json.loads(session_data)
        return None

    def update_session_access(self, session_id: str):
        """⏰ Update session last accessed time"""
        session_data = self.get_session(session_id)
        if session_data:
            session_data["last_accessed"] = datetime.utcnow().isoformat()
            ttl = self.redis_client.ttl(f"session:{session_id}")
            self.redis_client.setex(
                f"session:{session_id}",
                ttl,
                json.dumps(session_data)
            )

    def revoke_session(self, session_id: str):
        """🚫 Revoke user session"""
        session_data = self.get_session(session_id)
        if session_data:
            # 🗑️ Remove session
            self.redis_client.delete(f"session:{session_id}")

            # 🗂️ Remove from user sessions
            user_id = session_data["user_id"]
            self.redis_client.srem(f"user_sessions:{user_id}", session_id)
```

```
            logger.info(f"🚫 Session revoked: {session_id}")

    def revoke_all_user_sessions(self, user_id: int):
        """🚫 Revoke all sessions for a user"""
        user_sessions_key = f"user_sessions:{user_id}"
        sessions = self.redis_client.smembers(user_sessions_key)

        for session_id in sessions:
            self.revoke_session(session_id)

        logger.info(f"🚫 All sessions revoked for user {user_id}")

    def is_token_blacklisted(self, token: str) -> bool:
        """🚫 Check if token is blacklisted"""
        return self.redis_client.exists(f"blacklist:{token}") > 0

    def blacklist_token(self, token: str, expiry: Optional[timedelta] = None):
        """🚫 Add token to blacklist"""
        expiry = expiry or self.token_expiry
        self.redis_client.setex(
            f"blacklist:{token}",
            int(expiry.total_seconds()),
            "blacklisted"
        )
        logger.info("🚫 Token blacklisted")

# 🌐 Global authentication service instance
auth_service = AuthenticationService()
```

📧 Email Service Class

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.base import MIMEBase
from email import encoders
from typing import List, Dict, Any, Optional
import aiofiles
import aiosmtplib

class EmailService:
    """
    📧 Advanced email service with templates and attachments

    Features:
    📨 HTML/Plain text emails
    📎 File attachments
    🎨 Email templates
```

```
    📊 Delivery tracking
    🔄 Retry logic
    📱 Mobile-friendly emails
    """

    def __init__(self):
        """🏗️ Initialize email service"""
        self.smtp_server = os.getenv("SMTP_SERVER", "smtp.gmail.com")
        self.smtp_port = int(os.getenv("SMTP_PORT", 587))
        self.smtp_username = os.getenv("SMTP_USERNAME", "")
        self.smtp_password = os.getenv("SMTP_PASSWORD", "")
        self.from_email = os.getenv("FROM_EMAIL", "noreply@example.com")
        self.from_name = os.getenv("FROM_NAME", "FastAPI App")

        # 📊 Email tracking
        self.sent_emails = 0
        self.failed_emails = 0

        logger.info("📧 Email service initialized")

    async def __call__(self) -> "EmailService":
        """🎯 Dependency injection entry point"""
        # Test SMTP connection
        try:
            await self._test_connection()
        except Exception as e:
            logger.error(f"❌ SMTP connection test failed: {e}")
            # Don't raise exception - allow service to work in degraded mode

        return self

    async def _test_connection(self):
        """🔍 Test SMTP connection"""
        smtp = aiosmtplib.SMTP(hostname=self.smtp_server, port=self.smtp_port)
        await smtp.connect()
        await smtp.starttls()
        await smtp.login(self.smtp_username, self.smtp_password)
        await smtp.quit()
        logger.info("✅ SMTP connection test successful")

    async def send_email(
        self,
        to_emails: List[str],
        subject: str,
        body_text: str,
        body_html: Optional[str] = None,
        attachments: Optional[List[str]] = None,
        reply_to: Optional[str] = None
    ) -> Dict[str, Any]:
        """
        📧 Send email with advanced features
```

```
        Use Cases:
        📧 User notifications
        📄 Report delivery
        🛒 Order confirmations
        🚨 Alert notifications
        📰 Newsletter campaigns
        """

        try:
            # 📝 Create message
            msg = MIMEMultipart("alternative")
            msg["Subject"] = subject
            msg["From"] = f"{self.from_name} <{self.from_email}>"
            msg["To"] = ", ".join(to_emails)

            if reply_to:
                msg["Reply-To"] = reply_to

            # 📄 Add text content
            text_part = MIMEText(body_text, "plain", "utf-8")
            msg.attach(text_part)

            # 🎨 Add HTML content if provided
            if body_html:
                html_part = MIMEText(body_html, "html", "utf-8")
                msg.attach(html_part)

            # 📎 Add attachments if provided
            if attachments:
                for file_path in attachments:
                    await self._add_attachment(msg, file_path)

            # 📧 Send email
            smtp = aiosmtplib.SMTP(hostname=self.smtp_server, port=self.smtp_port)
            await smtp.connect()
            await smtp.starttls()
            await smtp.login(self.smtp_username, self.smtp_password)

            await smtp.send_message(msg)
            await smtp.quit()

            # 📊 Update statistics
            self.sent_emails += 1

            logger.info(f"📧 Email sent successfully to {len(to_emails)}
recipients")

            return {
                "status": "sent",
                "recipients": to_emails,
```

```
                "subject": subject,
                "sent_at": datetime.utcnow().isoformat(),
                "message_id": msg.get("Message-ID", "unknown")
            }

        except Exception as e:
            self.failed_emails += 1
            logger.error(f"❌ Email sending failed: {e}")

            return {
                "status": "failed",
                "recipients": to_emails,
                "subject": subject,
                "error": str(e),
                "failed_at": datetime.utcnow().isoformat()
            }

    async def _add_attachment(self, msg: MIMEMultipart, file_path: str):
        """📎 Add file attachment to email"""
        try:
            async with aiofiles.open(file_path, "rb") as f:
                attachment_data = await f.read()

            part = MIMEBase("application", "octet-stream")
            part.set_payload(attachment_data)
            encoders.encode_base64(part)

            filename = os.path.basename(file_path)
            part.add_header(
                "Content-Disposition",
                f"attachment; filename= {filename}"
            )

            msg.attach(part)
            logger.debug(f"📎 Added attachment: {filename}")

        except Exception as e:
            logger.error(f"❌ Failed to add attachment {file_path}: {e}")

    async def send_template_email(
        self,
        to_emails: List[str],
        template_name: str,
        template_data: Dict[str, Any],
        subject: Optional[str] = None
    ) -> Dict[str, Any]:
        """
        🎨 Send email using template

        Use Cases:
        👋 Welcome emails
```

```
        📄 Invoice/receipt emails
        🔔 Notification emails
        📊 Report emails
        """

        try:
            # 🎨 Load and render template
            template_content = await self._render_template(template_name,
template_data)

            # 📧 Send email
            return await self.send_email(
                to_emails=to_emails,
                subject=subject or template_content.get("subject", "No Subject"),
                body_text=template_content.get("text", ""),
                body_html=template_content.get("html", "")
            )

        except Exception as e:
            logger.error(f"❌ Template email failed: {e}")
            return {
                "status": "failed",
                "error": str(e),
                "template": template_name
            }

    async def _render_template(
        self,
        template_name: str,
        data: Dict[str, Any]
    ) -> Dict[str, str]:
        """🎨 Render email template with data"""

        # 📁 Template paths
        template_dir = "templates/emails"
        text_template_path = f"{template_dir}/{template_name}.txt"
        html_template_path = f"{template_dir}/{template_name}.html"

        result = {}

        # 📄 Load text template
        try:
            async with aiofiles.open(text_template_path, "r") as f:
                text_template = await f.read()
                result["text"] = text_template.format(**data)
        except FileNotFoundError:
            logger.warning(f"📄 Text template not found: {text_template_path}")

        # 🎨 Load HTML template
        try:
            async with aiofiles.open(html_template_path, "r") as f:
```

```
                html_template = await f.read()
                result["html"] = html_template.format(**data)
        except FileNotFoundError:
            logger.warning(f"🎨 HTML template not found: {html_template_path}")

        return result

    def get_statistics(self) -> Dict[str, Any]:
        """📊 Get email service statistics"""
        total_emails = self.sent_emails + self.failed_emails
        success_rate = (self.sent_emails / total_emails * 100) if total_emails > 0
else 0

        return {
            "sent_emails": self.sent_emails,
            "failed_emails": self.failed_emails,
            "total_emails": total_emails,
            "success_rate": round(success_rate, 2)
        }

# 🌐 Global email service instance
email_service = EmailService()
```

🎯 Class Dependencies Usage Examples

```
# 🎯 Endpoints using class dependencies

@app.post("/register")
async def register_user(
    user_data: dict,
    db: DatabaseService = Depends(database_service),
    auth: AuthenticationService = Depends(auth_service),
    email: EmailService = Depends(email_service)
):
    """
    👤 Register new user with multiple class dependencies

    Dependencies Used:
    💾 Database Service - Store user data
    🔐 Authentication Service - Create session
    📧 Email Service - Send welcome email
    """

    try:
        # 📝 Hash password
        hashed_password = get_password_hash(user_data["password"])

        # 👤 Create user in database
        new_user = await db.create_user({
```

```
            "username": user_data["username"],
            "email": user_data["email"],
            "hashed_password": hashed_password
        })

        # 🎫 Create authentication session
        session = auth.create_session(new_user["id"], new_user["username"])

        # 📧 Send welcome email
        email_result = await email.send_template_email(
            to_emails=[new_user["email"]],
            template_name="welcome",
            template_data={
                "username": new_user["username"],
                "login_url": "https://app.example.com/login"
            },
            subject=f"👋 Welcome to FastAPI App, {new_user['username']}!"
        )

        return {
            "message": "✅ User registered successfully",
            "user": {
                "id": new_user["id"],
                "username": new_user["username"],
                "email": new_user["email"]
            },
            "session": session,
            "email_sent": email_result["status"] == "sent"
        }

    except Exception as e:
        logger.error(f"❌ Registration failed: {e}")
        raise HTTPException(status_code=400, detail=str(e))

@app.get("/dashboard")
async def get_dashboard_data(
    current_user: User = Depends(get_current_active_user),
    db: DatabaseService = Depends(database_service),
    pagination: Dict[str, int] = Depends(get_pagination_params)
):
    """
    📊 Get dashboard data with authentication and database

    Dependencies Used:
    🔐 User Authentication - Verify access
    💾 Database Service - Fetch data
    📄 Pagination - Limit results
    """

    # 📊 Get user's recent activity
    user_activities = await db.execute_query(
```

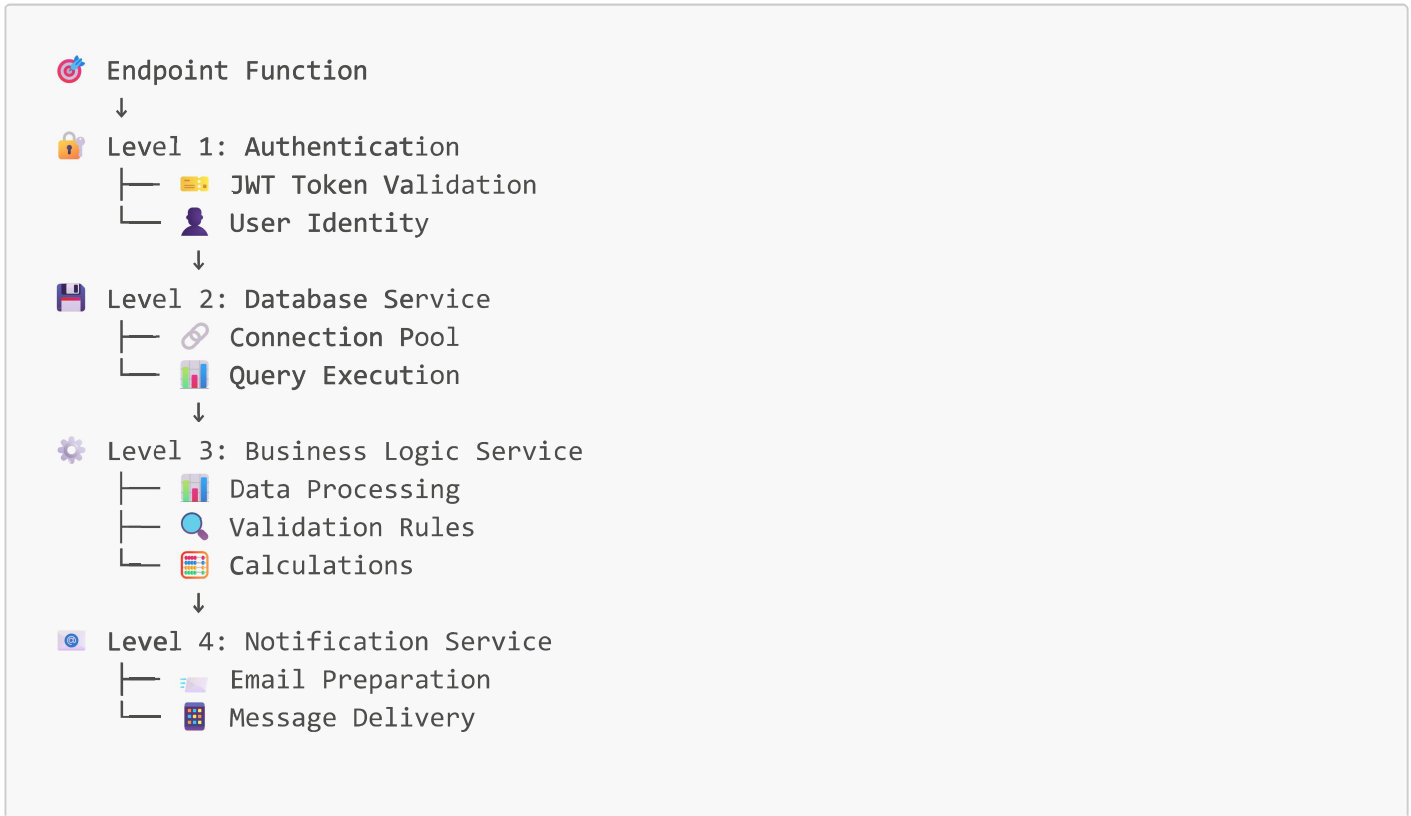
```
        "SELECT * FROM user_activities WHERE user_id = $1 ORDER BY created_at DESC
LIMIT $2 OFFSET $3",
        current_user.id,
        pagination["size"],
        pagination["offset"],
        fetch_all=True
    )

    # 📈 Get user statistics
    stats = await db.execute_query(
        "SELECT COUNT(*) as total_actions FROM user_activities WHERE user_id = $1",
        current_user.id,
        fetch_one=True
    )

    return {
        "user": {
            "id": current_user.id,
            "username": current_user.username
        },
        "activities": [dict(activity) for activity in user_activities],
        "statistics": dict(stats),
        "pagination": pagination
    }
```

🧬 Section 3: Multi-Level Dependencies

📊 Multi-Level Dependency Architecture



```
        ↓
🎯 Final Response
```

🛠️ Advanced Multi-Level Dependency Implementation

```python
from typing import List, Dict, Any, Optional
from decimal import Decimal
import asyncio

class OrderService:
    """
    🛒 Order management service with complex dependencies

    Multi-Level Dependencies:
    🔐 Authentication → 💾 Database → 📧 Email → 📊 Analytics
    """

    def __init__(
        self,
        db: DatabaseService,
        email: EmailService,
        auth: AuthenticationService
    ):
        """🏗️ Initialize with injected dependencies"""
        self.db = db
        self.email = email
        self.auth = auth

        # 🛒 Order configuration
        self.tax_rate = Decimal("0.08")  # 8% tax
        self.shipping_cost = Decimal("5.99")
        self.free_shipping_threshold = Decimal("50.00")

        logger.info("🛒 Order service initialized")

    async def __call__(self) -> "OrderService":
        """🎯 Dependency injection entry point"""
        return self

    async def create_order(
        self,
        user: User,
        items: List[Dict[str, Any]],
        shipping_address: Dict[str, str]
    ) -> Dict[str, Any]:
        """
        🛒 Create new order with full dependency chain

        Dependency Flow:
```

```
        👤 User (from auth) → 💾 Database operations → 📧 Email notifications
        """

        try:
            # 📊 Calculate order totals
            order_details = await self._calculate_order_totals(items)

            # 💾 Create order in database
            order_data = {
                "user_id": user.id,
                "items": items,
                "subtotal": float(order_details["subtotal"]),
                "tax_amount": float(order_details["tax_amount"]),
                "shipping_cost": float(order_details["shipping_cost"]),
                "total_amount": float(order_details["total_amount"]),
                "shipping_address": shipping_address,
                "status": "pending",
                "created_at": datetime.utcnow()
            }

            # 💾 Insert order into database
            order = await self.db.execute_query(
                """
                INSERT INTO orders (user_id, items, subtotal, tax_amount,
                                    shipping_cost, total_amount, shipping_address,
                                    status, created_at)
                VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9)
                RETURNING *
                """,
                order_data["user_id"],
                json.dumps(order_data["items"]),
                order_data["subtotal"],
                order_data["tax_amount"],
                order_data["shipping_cost"],
                order_data["total_amount"],
                json.dumps(order_data["shipping_address"]),
                order_data["status"],
                order_data["created_at"],
                fetch_one=True
            )

            order_dict = dict(order)

            # 📧 Send order confirmation email
            await self._send_order_confirmation(user, order_dict)

            # 📊 Update user analytics
            await self._update_user_analytics(user.id, order_dict)

            logger.info(f"🛒 Order created successfully: {order_dict['id']}")
```

```
            return {
                "order_id": order_dict["id"],
                "status": "created",
                "order_details": order_dict,
                "message": "✅ Order created successfully"
            }

        except Exception as e:
            logger.error(f"❌ Order creation failed: {e}")
            raise HTTPException(
                status_code=500,
                detail="🚫 Failed to create order"
            )

    async def _calculate_order_totals(self, items: List[Dict[str, Any]]) ->
Dict[str, Decimal]:
        """💰 Calculate order totals with tax and shipping"""

        subtotal = Decimal("0.00")

        # 🧮 Calculate subtotal
        for item in items:
            price = Decimal(str(item["price"]))
            quantity = item["quantity"]
            subtotal += price * quantity

        # 💸 Calculate tax
        tax_amount = subtotal * self.tax_rate

        # 🚚 Calculate shipping
        shipping_cost = Decimal("0.00") if subtotal >= self.free_shipping_threshold
else self.shipping_cost

        # 💰 Calculate total
        total_amount = subtotal + tax_amount + shipping_cost

        return {
            "subtotal": subtotal,
            "tax_amount": tax_amount,
            "shipping_cost": shipping_cost,
            "total_amount": total_amount
        }

    async def _send_order_confirmation(self, user: User, order: Dict[str, Any]):
        """📧 Send order confirmation email"""

        try:
            # 🎨 Prepare email template data
            template_data = {
                "username": user.username,
                "order_id": order["id"],
```

```
                "subtotal": f"${order['subtotal']:.2f}",
                "tax_amount": f"${order['tax_amount']:.2f}",
                "shipping_cost": f"${order['shipping_cost']:.2f}",
                "total_amount": f"${order['total_amount']:.2f}",
                "order_date": order["created_at"],
                "tracking_url": f"https://app.example.com/orders/{order['id']}"
            }

            # 📧 Send confirmation email
            await self.email.send_template_email(
                to_emails=[user.email],
                template_name="order_confirmation",
                template_data=template_data,
                subject=f"🛒 Order Confirmation #{order['id']}"
            )

            logger.info(f"📧 Order confirmation sent to {user.email}")

        except Exception as e:
            logger.error(f"❌ Failed to send order confirmation: {e}")
            # Don't fail the order creation if email fails

    async def _update_user_analytics(self, user_id: int, order: Dict[str, Any]):
        """📊 Update user analytics after order"""

        try:
            # 📈 Update user order statistics
            await self.db.execute_query(
                """
                INSERT INTO user_analytics (user_id, metric_type, metric_value,
recorded_at)
                VALUES ($1, $2, $3, $4)
                """,
                user_id,
                "order_created",
                order["total_amount"],
                datetime.utcnow()
            )

            logger.info(f"📊 Analytics updated for user {user_id}")

        except Exception as e:
            logger.error(f"❌ Failed to update analytics: {e}")
            # Don't fail the order if analytics update fails

class InventoryService:
    """
    📦 Inventory management with database dependency
    """

    def __init__(self, db: DatabaseService):
```

```
        """🏗️ Initialize with database dependency"""
        self.db = db
        logger.info("📦 Inventory service initialized")

    async def __call__(self) -> "InventoryService":
        """🎯 Dependency injection entry point"""
        return self

    async def check_availability(self, items: List[Dict[str, Any]]) -> Dict[str,
Any]:
        """📦 Check item availability"""

        availability_results = []
        all_available = True

        for item in items:
            product_id = item["product_id"]
            requested_quantity = item["quantity"]

            # 📊 Get current stock
            stock_info = await self.db.execute_query(
                "SELECT quantity, reserved_quantity FROM inventory WHERE product_id
= $1",
                product_id,
                fetch_one=True
            )

            if stock_info:
                available_quantity = stock_info["quantity"] -
stock_info["reserved_quantity"]
                is_available = available_quantity >= requested_quantity

                availability_results.append({
                    "product_id": product_id,
                    "requested_quantity": requested_quantity,
                    "available_quantity": available_quantity,
                    "is_available": is_available
                })

                if not is_available:
                    all_available = False
            else:
                # Product not found in inventory
                availability_results.append({
                    "product_id": product_id,
                    "requested_quantity": requested_quantity,
                    "available_quantity": 0,
                    "is_available": False
                })
                all_available = False
```

```
        return {
            "all_available": all_available,
            "items": availability_results
        }

    async def reserve_items(self, items: List[Dict[str, Any]]) -> bool:
        """📦 Reserve items for order"""

        try:
            # 🔄 Start database transaction
            async with self.db.get_connection() as conn:
                async with conn.transaction():
                    for item in items:
                        await conn.execute(
                            """
                            UPDATE inventory
                            SET reserved_quantity = reserved_quantity + $1
                            WHERE product_id = $2
                            """,
                            item["quantity"],
                            item["product_id"]
                        )

            logger.info(f"📦 Reserved {len(items)} items")
            return True

        except Exception as e:
            logger.error(f"❌ Failed to reserve items: {e}")
            return False

# 🎯 Complex dependency function that combines multiple services
async def get_order_service(
    db: DatabaseService = Depends(database_service),
    email: EmailService = Depends(email_service),
    auth: AuthenticationService = Depends(auth_service)
) -> OrderService:
    """
    🏗️ Multi-level dependency injection for OrderService

    Dependency Chain:
    🔐 Auth Service → 💾 Database Service → 📧 Email Service → 🛒 Order Service
    """

    # 🏗️ Create order service with all dependencies
    order_service = OrderService(db, email, auth)
    return await order_service()

async def get_inventory_service(
    db: DatabaseService = Depends(database_service)
) -> InventoryService:
    """
```

```
    🏗️ Dependency injection for InventoryService

    Dependency Chain:
    💾 Database Service → 📦 Inventory Service
    """

    inventory_service = InventoryService(db)
    return await inventory_service()

# 🎯 Complex endpoint with multi-level dependencies
@app.post("/orders")
async def create_order_endpoint(
    order_request: dict,
    current_user: User = Depends(get_current_active_user),
    order_service: OrderService = Depends(get_order_service),
    inventory_service: InventoryService = Depends(get_inventory_service)
):
    """
    🛒 Create order with complex multi-level dependencies

    Complete Dependency Chain:
    🎫 JWT Token → 🔐 Auth Service → 👤 Current User
    💾 Database Service → 🛒 Order Service
    💾 Database Service → 📦 Inventory Service
    📧 Email Service → 🛒 Order Service
    """

    try:
        items = order_request["items"]
        shipping_address = order_request["shipping_address"]

        # 📦 Check inventory availability
        availability = await inventory_service.check_availability(items)

        if not availability["all_available"]:
            unavailable_items = [
                item for item in availability["items"]
                if not item["is_available"]
            ]

            return {
                "error": "❌ Some items are not available",
                "unavailable_items": unavailable_items,
                "status": "inventory_check_failed"
            }

        # 📦 Reserve inventory
        reservation_success = await inventory_service.reserve_items(items)
        if not reservation_success:
            raise HTTPException(
                status_code=500,
```

```
                detail="🚫 Failed to reserve inventory"
            )

        # 🛒 Create order
        order_result = await order_service.create_order(
            user=current_user,
            items=items,
            shipping_address=shipping_address
        )

        return order_result

    except Exception as e:
        logger.error(f"❌ Order creation endpoint failed: {e}")
        raise HTTPException(
            status_code=500,
            detail="🚫 Failed to create order"
        )

@app.get("/orders/{order_id}")
async def get_order_details(
    order_id: int,
    current_user: User = Depends(get_current_active_user),
    db: DatabaseService = Depends(database_service)
):
    """
    🔍 Get order details with authentication and database dependencies

    Dependency Chain:
    🎫 JWT Token → 🔐 User Auth → 💾 Database Query
    """

    # 📊 Get order from database
    order = await db.execute_query(
        "SELECT * FROM orders WHERE id = $1 AND user_id = $2",
        order_id,
        current_user.id,
        fetch_one=True
    )

    if not order:
        raise HTTPException(
            status_code=404,
            detail="🔍 Order not found"
        )

    order_dict = dict(order)

    # 📊 Parse JSON fields
    order_dict["items"] = json.loads(order_dict["items"])
    order_dict["shipping_address"] = json.loads(order_dict["shipping_address"])
```

```
    return {
        "order": order_dict,
        "user": {
            "id": current_user.id,
            "username": current_user.username
        }
    }
```

📊 Multi-Level Dependencies Use Cases

Dependency Level	🎯 Service	🔗 Depends On	📊 Use Case
Level 1	Authentication	JWT Token	🔐 User identity verification
Level 2	Database	Connection Pool	💾 Data persistence
Level 3	Business Logic	Auth + Database	🧮 Core application logic
Level 4	Notifications	Email Service	📧 User communications
Level 5	Analytics	All above	📊 Data insights and tracking

🌐 Section 4: Global Dependencies

🏗️ Global Dependencies Architecture

```
📱 FastAPI Application
├── 🌐 App-Level Global Dependencies
│   ├── 🔒 Global Authentication
│   ├── 📝 Request Logging
│   ├── 🚦 Rate Limiting
│   └── 🛡️ Security Headers
│
├── 🎯 Router-Level Dependencies
│   ├── 📊 Admin Router → 👑 Admin Auth
│   ├── 🛒 API Router → 🔐 API Key Auth
│   └── 🎨 Public Router → 📝 Basic Logging
│
└── 🔗 Endpoint-Specific Dependencies
    ├── 💾 Database connections
    ├── 📧 Email services
    └── 🔍 Specific validations
```

🛠️ Global Dependencies Implementation

🔒 Global Security and Monitoring

```python
import time
from starlette.middleware.base import BaseHTTPMiddleware
from typing import Callable

# 🌐 Global dependency functions

async def global_request_logger(request: Request) -> Dict[str, Any]:
    """
    📝 Global request logging for all endpoints

    Applied to: ALL endpoints in the application
    Use Cases:
    📊 Request analytics
    🔍 Debugging and troubleshooting
    📈 Performance monitoring
    🚨 Security auditing
    """

    start_time = time.time()
    request_id = str(uuid.uuid4())

    # 📋 Extract request information
    request_info = {
        "request_id": request_id,
        "method": request.method,
        "url": str(request.url),
        "client_ip": request.client.host if request.client else "unknown",
        "user_agent": request.headers.get("user-agent", "unknown"),
        "start_time": start_time,
        "timestamp": datetime.utcnow().isoformat()
    }

    # 🗂️ Store in request state for access in endpoints
    request.state.request_info = request_info
    request.state.request_id = request_id
    request.state.start_time = start_time

    logger.info(f"📝 Request started: {request.method} {request.url.path} (ID:
{request_id})")

    return request_info

async def global_rate_limiter(request: Request) -> bool:
    """
    🚦 Global rate limiting for all endpoints

    Applied to: ALL endpoints in the application
    Use Cases:
    🛡️ DDoS protection
    🚫 API abuse prevention
```

```
    📊 Resource management
    ⚡ Performance optimization
    """

    client_ip = request.client.host if request.client else "unknown"

    # 🔑 Create rate limit key
    rate_limit_key = f"rate_limit:{client_ip}"

    # 📊 Check current request count (using Redis or in-memory cache)
    try:
        # Using Redis for distributed rate limiting
        import redis
        redis_client = redis.Redis(host="localhost", port=6379, db=1)

        current_requests = redis_client.get(rate_limit_key)
        if current_requests is None:
            # 🆕 First request from this IP
            redis_client.setex(rate_limit_key, 60, 1)  # 1 request in 60 seconds
window
            return True

        current_count = int(current_requests)
        max_requests = 100  # 100 requests per minute

        if current_count >= max_requests:
            logger.warning(f"🚦 Rate limit exceeded for IP: {client_ip}")
            raise HTTPException(
                status_code=429,
                detail="🚦 Too many requests. Please try again later.",
                headers={"Retry-After": "60"}
            )

        # 📈 Increment request count
        redis_client.incr(rate_limit_key)

        logger.debug(f"🚦 Rate limit check passed: {client_ip} ({current_count +
1}/{max_requests})")
        return True

    except redis.RedisError:
        # 🚨 If Redis is down, allow requests but log warning
        logger.warning("🚨 Rate limiting service unavailable - allowing request")
        return True

async def global_security_validator(request: Request) -> Dict[str, Any]:
    """
    🛡️ Global security validation for all requests

    Applied to: ALL endpoints in the application
    Use Cases:
```

```
    🚫 Malicious request detection
    🔒 Input sanitization
    🛡️ Header validation
    📊 Security analytics
    """

    security_info = {
        "is_secure": True,
        "warnings": [],
        "blocked": False
    }

    # 🔍 Check for suspicious patterns in URL
    suspicious_patterns = [
        "../", "..\\",  # Directory traversal
        "<script", "</script>",  # XSS attempts
        "union select", "drop table",  # SQL injection
        "eval(", "exec(",  # Code injection
    ]

    url_path = str(request.url.path).lower()
    query_string = str(request.url.query).lower()

    for pattern in suspicious_patterns:
        if pattern in url_path or pattern in query_string:
            security_info["warnings"].append(f"Suspicious pattern detected:
{pattern}")
            security_info["is_secure"] = False

            logger.warning(f"🚫 Suspicious request from {request.client.host}:
{pattern}")

    # 🔍 Validate headers
    user_agent = request.headers.get("user-agent", "")
    if len(user_agent) > 1000:  # Unusually long user agent
        security_info["warnings"].append("Unusually long User-Agent header")

    # 🚫 Block severely suspicious requests
    if len(security_info["warnings"]) >= 3:
        security_info["blocked"] = True
        raise HTTPException(
            status_code=400,
            detail="🚫 Request blocked due to security concerns"
        )

    # 🗂️ Store security info in request state
    request.state.security_info = security_info

    return security_info

async def global_performance_tracker(request: Request) -> Dict[str, Any]:
```

```
    """
    ⚡ Global performance tracking for all endpoints

    Applied to: ALL endpoints in the application
    Use Cases:
    📊 Performance monitoring
    🐌 Slow endpoint detection
    📈 Response time analytics
    🎯 Optimization insights
    """

    # 📊 Get memory usage before request
    import psutil
    process = psutil.Process()
    memory_before = process.memory_info().rss / 1024 / 1024  # MB

    performance_info = {
        "endpoint": f"{request.method} {request.url.path}",
        "memory_before": memory_before,
        "cpu_before": process.cpu_percent(),
        "start_time": time.time()
    }

    # 🗂️ Store in request state
    request.state.performance_info = performance_info

    return performance_info

# 🌐 Apply global dependencies to the entire app
app = FastAPI(
    title="🌐 Global Dependencies Demo",
    version="1.0.0",
    dependencies=[
        Depends(global_request_logger),
        Depends(global_rate_limiter),
        Depends(global_security_validator),
        Depends(global_performance_tracker)
    ]
)
```

🎯 Router-Level Dependencies

```
from fastapi import APIRouter

# 👑 Admin router with admin-only dependencies
admin_router = APIRouter(
    prefix="/admin",
    tags=["admin"],
```

```
    dependencies=[
        Depends(get_admin_user),  # Require admin authentication
        Depends(admin_audit_logger),  # Special logging for admin actions
        Depends(admin_rate_limiter)   # Different rate limits for admins
    ]
)

async def admin_audit_logger(
    request: Request,
    admin_user: User = Depends(get_admin_user)
) -> Dict[str, Any]:
    """
    📋 Special audit logging for admin actions

    Applied to: All admin endpoints
    Use Cases:
    🔍 Admin action tracking
    📊 Compliance and auditing
    🚨 Security monitoring
    📈 Admin usage analytics
    """

    audit_info = {
        "admin_user_id": admin_user.id,
        "admin_username": admin_user.username,
        "action": f"{request.method} {request.url.path}",
        "timestamp": datetime.utcnow().isoformat(),
        "ip_address": request.client.host if request.client else "unknown"
    }

    # 📝 Log admin action
    logger.info(f"👑 Admin action: {admin_user.username} performed
{audit_info['action']}")

    # 💾 Store in database for audit trail
    # await audit_db.log_admin_action(audit_info)

    request.state.audit_info = audit_info
    return audit_info

async def admin_rate_limiter(
    request: Request,
    admin_user: User = Depends(get_admin_user)
) -> bool:
    """
    🚦 Special rate limiting for admin users

    Applied to: All admin endpoints
    Different limits: Higher limits for admin users
    """
```

```python
    # 📊 Admins get higher rate limits
    rate_limit_key = f"admin_rate_limit:{admin_user.id}"
    max_requests = 500  # 500 requests per minute for admins

    # Implementation similar to global rate limiter but with higher limits
    logger.debug(f"🚦 Admin rate limit check for {admin_user.username}")
    return True

# 🛒 API router with API key authentication
api_router = APIRouter(
    prefix="/api/v1",
    tags=["api"],
    dependencies=[
        Depends(validate_api_key),      # Require valid API key
        Depends(api_usage_tracker),     # Track API usage
        Depends(api_rate_limiter)       # API-specific rate limiting
    ]
)

async def validate_api_key(api_key: str = Header(None, alias="X-API-Key")) ->
Dict[str, Any]:
    """
    🔑 Validate API key for API router

    Applied to: All /api/v1 endpoints
    Use Cases:
    🔐 Third-party integration authentication
    📊 API usage tracking
    💰 Billing and quota management
    🚫 Access control
    """

    if not api_key:
        raise HTTPException(
            status_code=401,
            detail="🔑 API key required",
            headers={"WWW-Authenticate": "ApiKey"}
        )

    # 🔍 Validate API key (check against database)
    # api_key_info = await db.get_api_key(api_key)

    # 🎭 Mock validation for demo
    valid_api_keys = {
        "demo-key-123": {
            "client_id": "demo-client",
            "client_name": "Demo Client",
            "quota_limit": 1000,
            "quota_used": 45
        }
    }
```

```
    api_key_info = valid_api_keys.get(api_key)
    if not api_key_info:
        logger.warning(f"🚫 Invalid API key attempted: {api_key[:10]}...")
        raise HTTPException(
            status_code=401,
            detail="🚫 Invalid API key"
        )

    # 📊 Check quota
    if api_key_info["quota_used"] >= api_key_info["quota_limit"]:
        raise HTTPException(
            status_code=429,
            detail="📊 API quota exceeded"
        )

    logger.info(f"🔑 Valid API key: {api_key_info['client_name']}")

    return api_key_info

async def api_usage_tracker(
    request: Request,
    api_key_info: Dict[str, Any] = Depends(validate_api_key)
) -> Dict[str, Any]:
    """
    📊 Track API usage for billing and analytics

    Applied to: All /api/v1 endpoints
    Use Cases:
    💰 Billing calculations
    📈 Usage analytics
    📊 Quota management
    🎯 Performance insights
    """

    usage_info = {
        "client_id": api_key_info["client_id"],
        "client_name": api_key_info["client_name"],
        "endpoint": f"{request.method} {request.url.path}",
        "timestamp": datetime.utcnow().isoformat(),
        "quota_used": api_key_info["quota_used"] + 1,
        "quota_limit": api_key_info["quota_limit"]
    }

    # 📊 Update usage in database
    # await db.update_api_usage(api_key_info["client_id"])

    logger.info(f"📊 API usage: {usage_info['client_name']} -
{usage_info['endpoint']}")

    request.state.api_usage_info = usage_info
```

```
    return usage_info

async def api_rate_limiter(
    api_key_info: Dict[str, Any] = Depends(validate_api_key)
) -> bool:
    """
    🚦 API-specific rate limiting

    Applied to: All /api/v1 endpoints
    Different limits based on API plan
    """

    # 📊 Different rate limits based on client tier
    client_tier = "premium"  # This would come from API key info

    rate_limits = {
        "basic": 60,      # 60 requests per minute
        "premium": 300,   # 300 requests per minute
        "enterprise": 1000 # 1000 requests per minute
    }

    max_requests = rate_limits.get(client_tier, 60)

    logger.debug(f"🚦 API rate limit: {api_key_info['client_name']} ({client_tier}:
{max_requests}/min)")
    return True

# 🎨 Public router with minimal dependencies
public_router = APIRouter(
    prefix="/public",
    tags=["public"],
    dependencies=[
        Depends(public_access_logger)  # Basic logging only
    ]
)

async def public_access_logger(request: Request) -> Dict[str, Any]:
    """
    📝 Basic logging for public endpoints

    Applied to: All /public endpoints
    Use Cases:
    📊 Public API analytics
    🔍 Basic monitoring
    📈 Traffic analysis
    """

    access_info = {
        "endpoint": f"{request.method} {request.url.path}",
        "timestamp": datetime.utcnow().isoformat(),
        "ip": request.client.host if request.client else "unknown",
```

```
        "access_type": "public"
    }

    logger.info(f"🌐 Public access: {access_info['endpoint']}")

    request.state.public_access_info = access_info
    return access_info

# 🔗 Add routers to main app
app.include_router(admin_router)
app.include_router(api_router)
app.include_router(public_router)
```

🎯 Global Dependencies Usage Examples

```
# 👑 Admin endpoints (inherits admin dependencies)
@admin_router.get("/users")
async def admin_get_all_users(
    pagination: Dict[str, int] = Depends(get_pagination_params),
    db: DatabaseService = Depends(database_service)
):
    """
    👑 Admin endpoint to get all users

    Global Dependencies Applied:
    📝 Global request logging
    🚦 Global rate limiting
    🛡️ Global security validation
    ⚡ Global performance tracking

    Router Dependencies Applied:
    👑 Admin authentication
    📋 Admin audit logging
    🚦 Admin rate limiting
    """

    # 📊 Get all users with pagination
    users = await db.get_users_paginated(
        offset=pagination["offset"],
        limit=pagination["limit"]
    )

    return {
        "users": users,
        "pagination": pagination,
        "total_count": len(users)
    }
```

```python
@admin_router.post("/users/{user_id}/disable")
async def admin_disable_user(
    user_id: int,
    db: DatabaseService = Depends(database_service)
):
    """
    🚫 Admin endpoint to disable user

    All admin dependencies automatically applied
    """

    # 🔄 Update user status
    await db.execute_query(
        "UPDATE users SET is_active = false WHERE id = $1",
        user_id
    )

    return {"message": f"✅ User {user_id} disabled successfully"}

# 🔑 API endpoints (inherits API dependencies)
@api_router.get("/users")
async def api_get_users(
    pagination: Dict[str, int] = Depends(get_pagination_params),
    db: DatabaseService = Depends(database_service)
):
    """
    🔑 API endpoint to get users

    Global Dependencies Applied:
    📝 Request logging, 🚦 Rate limiting, 🛡️ Security validation, ⚡ Performance
tracking

    Router Dependencies Applied:
    🔑 API key validation, 📊 Usage tracking, 🚦 API rate limiting
    """

    users = await db.get_users_paginated(
        offset=pagination["offset"],
        limit=pagination["limit"]
    )

    return {
        "data": users,
        "pagination": pagination
    }

@api_router.post("/users")
async def api_create_user(
    user_data: dict,
    db: DatabaseService = Depends(database_service)
):
```

```
    """
    👤 API endpoint to create user

    All API dependencies automatically applied
    """

    # 👤 Create user
    new_user = await db.create_user(user_data)

    return {
        "message": "✅ User created successfully",
        "user": new_user
    }

# 🌐 Public endpoints (minimal dependencies)
@public_router.get("/health")
async def public_health_check():
    """
    🏥 Public health check endpoint

    Global Dependencies Applied:
    📝 Request logging, 🚦 Rate limiting, 🛡️ Security validation, ⚡ Performance
tracking

    Router Dependencies Applied:
    📝 Public access logging
    """

    return {
        "status": "🟢 healthy",
        "timestamp": datetime.utcnow().isoformat(),
        "version": "1.0.0"
    }

@public_router.get("/status")
async def public_system_status():
    """
    📊 Public system status endpoint

    Only public dependencies applied
    """

    return {
        "system": "🟢 operational",
        "database": "🟢 connected",
        "cache": "🟢 available",
        "last_updated": datetime.utcnow().isoformat()
    }

# 🎯 Endpoint showing all dependency info
@app.get("/debug/dependencies")
```

```python
async def debug_dependencies_info(
    request: Request,
    current_user: User = Depends(get_current_active_user)
):
    """
    🔍 Debug endpoint showing all applied dependencies

    Shows information from all global and router dependencies
    """

    # 📊 Collect dependency information from request state
    dependency_info = {
        "global_dependencies": {
            "request_info": getattr(request.state, "request_info", None),
            "security_info": getattr(request.state, "security_info", None),
            "performance_info": getattr(request.state, "performance_info", None)
        },
        "authentication": {
            "user_id": current_user.id,
            "username": current_user.username,
            "is_admin": current_user.is_admin
        },
        "endpoint_specific": {
            "endpoint": f"{request.method} {request.url.path}",
            "authenticated": True
        }
    }

    # 📋 Add router-specific dependency info if available
    if hasattr(request.state, "audit_info"):
        dependency_info["admin_dependencies"] = request.state.audit_info

    if hasattr(request.state, "api_usage_info"):
        dependency_info["api_dependencies"] = request.state.api_usage_info

    if hasattr(request.state, "public_access_info"):
        dependency_info["public_dependencies"] = request.state.public_access_info

    return dependency_info
```

📊 Global Dependencies Summary & Best Practices

Dependency Level	🎯 Scope	📝 Configuration	🛠️ Use Cases
🌐 **Application**	All endpoints	`FastAPI(dependencies=[...])`	📝 Logging, 🚦 Rate limiting, 🛡️ Security

Dependency Level	🎯 Scope	📝 Configuration	🛠️ Use Cases
🎯 **Router**	Router endpoints	`APIRouter(dependencies=[...])`	🔐 Auth types, 📊 Analytics, 🎯 Access control
🔗 **Endpoint**	Single endpoint	Function parameters	💾 Data access, 📧 Services, 🔍 Validation

🎯 Section 5: Best Practices & Production Guidelines

📋 Dependency Best Practices Checklist

✅ Design Principles

```python
# ✅ DO: Keep dependencies focused and single-purpose
async def get_database_connection() -> DatabaseConnection:
    """💾 Single responsibility: provide database connection"""
    return await create_connection()

# ❌ DON'T: Mix multiple concerns in one dependency
async def get_everything():
    """❌ Violates single responsibility principle"""
    db = await create_connection()
    user = await authenticate_user()
    email = await setup_email()
    return {"db": db, "user": user, "email": email}

# ✅ DO: Use type hints for better IDE support and documentation
async def get_user_service(
    db: DatabaseService = Depends(get_database)
) -> UserService:
    """👤 Type-annotated dependency for better tooling support"""
    return UserService(db)

# ✅ DO: Handle errors gracefully in dependencies
async def get_redis_cache() -> Optional[RedisCache]:
    """💾 Graceful error handling for optional services"""
    try:
        cache = RedisCache()
        await cache.ping()
        return cache
    except Exception as e:
        logger.warning(f"💾 Redis unavailable: {e}")
        return None  # Fallback to no caching

# ✅ DO: Use configuration-based dependencies
class AppConfig:
    """⚙️ Configuration class for dependency injection"""
```

```
    def __init__(self):
        self.database_url = os.getenv("DATABASE_URL")
        self.redis_url = os.getenv("REDIS_URL")
        self.email_backend = os.getenv("EMAIL_BACKEND", "smtp")

async def get_config() -> AppConfig:
    """⚙️ Configuration dependency"""
    return AppConfig()

async def get_database_with_config(
    config: AppConfig = Depends(get_config)
) -> DatabaseService:
    """💾 Database service with configuration dependency"""
    return DatabaseService(config.database_url)
```

🧪 Testing Dependencies

```
import pytest
from fastapi.testclient import TestClient
from unittest.mock import AsyncMock, MagicMock

# 🧪 Test database dependency override
async def get_test_database():
    """💾 Test database dependency override"""
    return MockDatabaseService()

class MockDatabaseService:
    """🎭 Mock database for testing"""

    async def get_user_by_id(self, user_id: int):
        return {
            "id": user_id,
            "username": f"test_user_{user_id}",
            "email": f"test{user_id}@example.com"
        }

    async def create_user(self, user_data: dict):
        return {
            "id": 999,
            **user_data,
            "created_at": "2024-01-01T00:00:00"
        }

# 🧪 Test authentication dependency override
async def get_test_user():
    """👤 Test user dependency override"""
    return User(
        id=1,
```

```python
        username="test_user",
        email="test@example.com",
        is_active=True,
        is_admin=False,
        created_at=datetime.utcnow()
    )

async def get_test_admin_user():
    """👑 Test admin user dependency override"""
    return User(
        id=1,
        username="test_admin",
        email="admin@example.com",
        is_active=True,
        is_admin=True,
        created_at=datetime.utcnow()
    )

# 🧪 Test setup with dependency overrides
@pytest.fixture
def test_client():
    """🧪 Test client with dependency overrides"""

    # Override dependencies for testing
    app.dependency_overrides[database_service] = get_test_database
    app.dependency_overrides[get_current_active_user] = get_test_user

    client = TestClient(app)
    yield client

    # Clean up overrides
    app.dependency_overrides.clear()

@pytest.fixture
def admin_test_client():
    """👑 Test client with admin user"""

    app.dependency_overrides[database_service] = get_test_database
    app.dependency_overrides[get_current_active_user] = get_test_admin_user

    client = TestClient(app)
    yield client

    app.dependency_overrides.clear()

# 🧪 Example test cases
def test_create_user_endpoint(test_client):
    """🧪 Test user creation with mocked dependencies"""

    user_data = {
        "username": "new_user",
```

```
        "email": "new@example.com",
        "password": "password123"
    }

    response = test_client.post("/users", json=user_data)

    assert response.status_code == 200
    data = response.json()
    assert data["user"]["username"] == "new_user"
    assert "password" not in data["user"]  # Ensure password not returned

def test_admin_endpoint_access(admin_test_client):
    """👑 Test admin endpoint with admin user"""

    response = admin_test_client.get("/admin/users")

    assert response.status_code == 200
    data = response.json()
    assert "users" in data

def test_admin_endpoint_forbidden(test_client):
    """🚫 Test admin endpoint with regular user"""

    response = test_client.get("/admin/users")

    assert response.status_code == 403
    assert "Insufficient permissions" in response.json()["detail"]
```

🚀 Performance Optimization

```
from functools import lru_cache
import asyncio
from typing import Dict, Any

# 🚀 Cached dependencies for expensive operations
@lru_cache(maxsize=1)
def get_app_settings() -> Dict[str, Any]:
    """⚙️ Cached application settings (computed once)"""

    # Expensive configuration loading
    settings = {
        "database_pool_size": int(os.getenv("DB_POOL_SIZE", 10)),
        "cache_ttl": int(os.getenv("CACHE_TTL", 3600)),
        "feature_flags": load_feature_flags(),  # Expensive operation
        "external_api_config": load_api_configs()  # Network calls
    }

    logger.info("⚙️ Application settings loaded and cached")
```

```python
    return settings

# 🔄 Connection pooling for database dependencies
class DatabaseConnectionPool:
    """💾 Database connection pool for performance"""

    def __init__(self):
        self._pool = None
        self._lock = asyncio.Lock()

    async def get_pool(self):
        """🔗 Get or create connection pool"""
        if self._pool is None:
            async with self._lock:
                if self._pool is None:  # Double-check locking
                    self._pool = await asyncpg.create_pool(
                        dsn=os.getenv("DATABASE_URL"),
                        min_size=5,
                        max_size=20,
                        command_timeout=30
                    )
                    logger.info("💾 Database pool created")

        return self._pool

# 🌐 Global connection pool instance
db_pool = DatabaseConnectionPool()

async def get_database_connection():
    """💾 Get database connection from pool"""
    pool = await db_pool.get_pool()
    async with pool.acquire() as connection:
        yield connection

# ⚡ Async dependency with connection reuse
class OptimizedUserService:
    """👤 Optimized user service with connection reuse"""

    def __init__(self, db_connection):
        self.db = db_connection
        self._user```
```