Immagine che contiene testo

Descrizione generata automaticamente

**Hyperautomation Development Guidelines**

Documento Confidenziale

[1. Architecture & Design Principles 5](#_Toc187745812)

[1.1. Modular & Layered Architecture 5](#_Toc187745813)

[1.2. API-Driven Communication 5](#_Toc187745814)

[1.3. Data Management 5](#_Toc187745815)

[1.4. Security by Design 5](#_Toc187745816)

[1.5. Performance & Scalability 5](#_Toc187745817)

[2. Coding Standards & Practices 6](#_Toc187745818)

[2.1. UiPath (RPA) 6](#_Toc187745819)

[2.1.1. Project Organization & Structure 6](#_Toc187745820)

[2.1.2. Naming Conventions 6](#_Toc187745821)

[2.1.3. Best Practices & Coding Standards 7](#_Toc187745822)

[2.1.4. 2.4 Error Handling & Logging 8](#_Toc187745823)

[2.1.5. Version Control & Branching 9](#_Toc187745824)

[2.1.6. Testing & Debugging 9](#_Toc187745825)

[2.1.7. Deployment & Orchestrator 10](#_Toc187745826)

[2.1.8. Security & Compliance 10](#_Toc187745827)

[2.1.9. Performance & Scalability 11](#_Toc187745828)

[2.1.10. Documentation & Knowledge Sharing 11](#_Toc187745829)

[2.2. Appsmith (Low-Code) 12](#_Toc187745830)

[2.2.1. Application Structure 12](#_Toc187745831)

[2.2.2. Data Integration & APIs 12](#_Toc187745832)

[2.2.3. UI/UX Consistency 12](#_Toc187745833)

[2.2.4. Security 12](#_Toc187745834)

[2.3. React (Front-End) 12](#_Toc187745835)

[2.3.1. Project Structure 12](#_Toc187745836)

[2.3.2. State Management 12](#_Toc187745837)

[2.3.3. Code Quality 13](#_Toc187745838)

[2.3.4. Testing 13](#_Toc187745839)

[2.3.5. Security & Performance 13](#_Toc187745840)

[2.4. Python FastAPI (Back-End) 13](#_Toc187745841)

[2.4.1. 2.4.1 Project Layout & Structure 13](#_Toc187745842)

[2.4.2. Endpoints & Routing 16](#_Toc187745843)

[2.4.3. Pydantic Models & Data Validation 17](#_Toc187745844)

[2.4.4. Database Integration 17](#_Toc187745845)

[2.4.5. Business Logic & Services 20](#_Toc187745846)

[2.4.6. Authentication & Authorization 21](#_Toc187745847)

[2.4.7. Error Handling & Exception Management 22](#_Toc187745848)

[2.4.8. Testing 23](#_Toc187745849)

[2.4.9. Security & Performance 24](#_Toc187745850)

[2.4.10. Logging & Monitoring 25](#_Toc187745851)

[2.4.11. Deployment & CI/CD 26](#_Toc187745852)

[2.4.12. Advanced Topics & Recommendations 26](#_Toc187745853)

[2.5. PostgreSQL / Database (Updated) 27](#_Toc187745854)

[2.5.1. Context & SQL Engine 27](#_Toc187745855)

[2.5.2. Recommended Software 27](#_Toc187745856)

[2.5.3. Number of Databases & Naming 27](#_Toc187745857)

[2.5.4. Architectural Rules 28](#_Toc187745858)

[2.5.5. Schema Design & Versioning 28](#_Toc187745859)

[2.5.6. SQL Query Rules 28](#_Toc187745860)

[2.5.7. Indexing & Performance Tuning 28](#_Toc187745861)

[2.5.8. Backup & Logging 29](#_Toc187745862)

[2.6. Blob / S3 File Server 29](#_Toc187745863)

[2.6.1. File Handling 29](#_Toc187745864)

[2.6.2. Access Control 29](#_Toc187745865)

[2.6.3. Lifecycle Policies 29](#_Toc187745866)

[3. Deployment & Infrastructure 29](#_Toc187745867)

[3.1. Containerization 29](#_Toc187745868)

[3.2. Nginx Configuration 29](#_Toc187745869)

[3.3. CI/CD Pipeline 30](#_Toc187745870)

[3.4. Infrastructure as Code 30](#_Toc187745871)

[3.5. Environment Configuration 30](#_Toc187745872)

[4. Quality Assurance & Testing 30](#_Toc187745873)

[4.1. Test Strategy 30](#_Toc187745874)

[4.2. Code Reviews 30](#_Toc187745875)

[4.3. Automated Testing in CI 30](#_Toc187745876)

[4.4. Performance Testing 30](#_Toc187745877)

[4.5. Security Testing 31](#_Toc187745878)

[5. 5. Monitoring & Logging 31](#_Toc187745879)

[5.1. Logging 31](#_Toc187745880)

[5.2. Metrics & Observability 31](#_Toc187745881)

[5.3. Tracing 31](#_Toc187745882)

[5.4. Incident Response 31](#_Toc187745883)

[6. Governance & Compliance 31](#_Toc187745884)

[6.1. Policy Adherence 31](#_Toc187745885)

[6.2. Roles & Responsibilitie 31](#_Toc187745886)

[6.3. Regulatory Compliance 32](#_Toc187745887)

[6.4. Change Management 32](#_Toc187745888)

[7. Documentation & Knowledge Sharing 32](#_Toc187745889)

[7.1. Project README 32](#_Toc187745890)

[7.2. API Documentation 32](#_Toc187745891)

[7.3. User Guides & Tutorial 32](#_Toc187745892)

[7.4. Onboarding & Training 32](#_Toc187745893)

# Architecture & Design Principles

## Modular & Layered Architecture

* + Keep solutions modular with clear boundaries (e.g., separate RPA workflows from custom apps, separate front-end from back-end).
  + Use microservices or service-oriented design for custom back-end solutions so you can scale specific components independently.

## API-Driven Communication

* + Favor RESTful (FastAPI) or RPC endpoints, ensuring versioning (e.g., /v1, /v2) for backward compatibility.
  + Document APIs (e.g., with OpenAPI/Swagger) so consumers (Appsmith, React) understand request/response structures.

## Data Management

* + Use PostgreSQL as the primary relational DB.
  + Store unstructured data or large files in a blob/S3 file server.
  + Enforce well-defined data models and relationships to avoid messy joins or data redundancies.
  + Follow naming conventions (snake\_case for columns, singular or plural table names consistently).

## Security by Design

* + Define a clear authentication & authorization model (JWT or OAuth2 in FastAPI).
  + Encrypt data in transit (HTTPS via Nginx termination) and at rest (where feasible in S3 and Postgres).
  + Perform threat modeling and adhere to the principle of least privilege for all modules.

## Performance & Scalability

* + Configure caching (using Redis or in-memory caches) for frequently accessed data.
  + Use asynchronous operations in FastAPI for I/O-bound tasks.
  + Scale horizontally (containerization, orchestrators) as traffic grows.

# Coding Standards & Practices

## UiPath (RPA)

Robotic Process Automation (RPA) with UiPath is a powerful way to automate repetitive tasks and integrate systems that do not have direct APIs or easy integration points. To ensure consistency, maintainability, and reliability, follow the guidelines below.

### Project Organization & Structure

#### Solution Folder Layout

* + Keep your UiPath solution in a clearly named repository or folder. If your automation scope is large, break it into multiple projects or subfolders to represent different business processes, each with its own Main.xaml.
  + Consider using the REFramework (Robotic Enterprise Framework) for complex processes that require robust transaction handling, logging, and recovery.
  + For simpler automations, you may use State Machine or Flowcharts, but still keep them organized in subfolders that group related activities (e.g., DataRetrieval, DataProcessing, Reporting).

#### XAML File Segmentation

* + Avoid one giant workflow. Split your process into smaller, focused workflows based on specific functionality (e.g., Login.xaml, ExtractData.xaml, ProcessInvoice.xaml).
  + Keep each workflow as self-contained as possible to encourage reuse and easiertesting**.**

#### Config File Usage

* + Centralize your configuration parameters in a Config.xlsx or a JSON file so that environment-specific data (paths, credentials, queue names, URLs) are not hardcoded.
  + Use the Orchestrator Assets feature for secrets or sensitive credentials, referencing them at runtime through the config.

#### Argument Management

* + Use descriptive names for arguments (e.g., in\_CustomerData, out\_ProcessedData).
  + Avoid passing unnecessary data; keep arguments minimal and relevant to the workflow.
  + Follow naming conventions, such as prefixing arguments with in\_, out\_, or io\_ to indicate direction.

### Naming Conventions

#### File & Workflow Names

* + Prefer descriptive workflow names: AuthenticateUser.xaml over WF1.xaml.
  + Follow a consistent capitalization scheme (PascalCase or similar).

#### Variables

* + Use camelCase for variables (e.g., customerName, invoiceList).
  + Keep variable scope as narrow as possible to avoid naming collisions and confusion.

#### Activities & Sequences

* + Label key activities within a sequence with meaningful Display Names (e.g., “Click ‘Submit’ Button”).
  + For repeated patterns, consider creating reusable snippets or templates.

### Best Practices & Coding Standards

#### Use of the REFramework

* + For enterprise-grade automation or complex processes, start with the UiPath REFramework as a template.
  + Configure the Init state to handle environment setup, credential retrieval, config loading.
  + Use the Get Transaction Data and Process Transaction states to handle queue items or iterative data sets.
  + Leverage built-in logging, exception handling, and retry mechanisms to increase resilience.

#### Modularity & Reusability

* + Extract repetitive tasks into Reusable Components or separate XAML files that can be invoked by multiple processes.
  + Publish commonly used components as Reusable Libraries in Orchestrator so other teams or processes can integrate them easily.

#### Selectors & UI Automation

* + Keep selectors as stable and dynamic as possible (avoid overly specific attributes like idx if you can).
  + Use wildcards where needed, but ensure they do not become too general.
  + Document any assumptions about the target environment (e.g., screen resolution, browser versions).

#### Handling Citrix/VDI Automation

* + If dealing with Citrix or virtual desktop interfaces, prefer Computer Vision activities for reliability.
  + If text-based automation is possible (e.g., OCR, hotkeys), evaluate these for potential performance and stability gains.

#### Orchestrator Queues

* + When working with multiple transactions, store them in Orchestrator Queues for better scalability and monitoring.
  + Use queue item references or custom progress statuses to track processing stages.
  + Handle business exceptions separately from system exceptions to allow for targeted retries or specific error handling.

#### Config and Credential Management

* + Never hardcode passwords or API keys in workflows.
  + Store sensitive data in Orchestrator Assets or integrate with a secure password vault.
  + Make sure your process gracefully handles missing or invalid credentials.

### 2.4 Error Handling & Logging

#### Structured Error Handling

* + Wrap critical steps in Try-Catch activities.
  + Distinguish between Business Exceptions (e.g., data validation issues) and System Exceptions (e.g., network errors, element not found).
  + Provide meaningful messages in Throw or Rethrow activities to aid troubleshooting.

#### Logging Standards

* + Leverage Log Message activities at key points: start/end of workflows, transitions, decision points, errors.
  + Use consistent log levels: Trace for very detailed logs, Info for normal events, Warn for recoverable issues, Error for failures.
  + Write logs to Orchestrator or a centralized log aggregation system (e.g., Splunk, Elastic Stack) for monitoring and analytics.

#### Exception Screenshots

* + For UI automations, capture screenshots on errors or exceptions.
  + Store these in a designated folder or push them to a secure location (e.g., an S3 bucket) for later review.

#### Retry Mechanisms

* + Use built-in retry strategies for queue transactions within the REFramework or Orchestrator.
  + For critical standalone workflows, add logic to handle transient issues (like a short network glitch).

### Version Control & Branching

#### Source Control Integration

* + Commit your entire UiPath project folder (including the .xaml files, .screenshots, project.json) to Git.
  + Exclude files that contain sensitive info or are environment-specific (e.g., a local Config.xlsx with secrets).

#### Branching Strategy

* + Use feature branches for new automation or major changes.
  + Merge to the main or master branch only after peer review and testing.
  + Tag releases in Git to match official Orchestrator package versions.

#### Continuous Integration

* + Automate a build process to create a NuGet package from the UiPath solution.
  + Store built packages in a shared feed (Orchestrator feed or internal registry) for standardized deployments.

### Testing & Debugging

#### Local Testing

* + Always test workflows locally with sample data before pushing to version control or Orchestrator.
  + Use Debug mode in UiPath Studio to step through workflows, set breakpoints, and watch variables.

#### Unit Testing

* + For complex logic or critical parts, create small, dedicated test workflows that verify functionality in isolation.
  + UiPath Testing capabilities (Test Manager, if available) can help systematically track coverage.

#### Integration Testing

* + If your automation depends on external systems (e.g., web services, databases), set up a staging environment or mocks.
  + Test the end-to-end flow, ensuring that each interface is validated.

#### Orchestrator Test Runs

* + Schedule test runs or use triggers in non-production Orchestrator environments.
  + Monitor logs and queue statuses to confirm correct operation at scale.

### Deployment & Orchestrator

#### Package Creation & Deployment

* + Standardize how packages are built (e.g., via UiPath Studio or a CI pipeline).
  + Include metadata (version, release notes, change log) when publishing to Orchestrator.

#### Environments & Tenants

* + Separate Development, Test/Staging, and Production environments (or Tenants) in Orchestrator.
  + Follow the principle of least privilege—only assign necessary permissions to each environment.

#### Robot Management

* + Keep your unattended and attended robots well-organized, with clear naming and descriptions.
  + Periodically review robot licenses to ensure compliance and cost-effectiveness.

#### Schedule Management

* + Use Orchestrator schedules for routine tasks; avoid relying on local cron jobs or manual triggers.
  + Stagger job schedules to prevent resource bottlenecks if multiple processes require the same environment.

### Security & Compliance

#### Credential Handling

* + Store credentials in Orchestrator Assets or a secure vault (e.g., CyberArk, Azure Key Vault).
  + Use encryption in transit (HTTPS) for all communications between UiPath Robots, Orchestrator, and external services.

#### Role-Based Access Control

* + Configure Orchestrator roles so that only authorized personnel can deploy or edit processes.
  + Limit who can view logs or handle credentials to reduce the risk of data breaches.

#### Auditing & Compliance

* + Enable Orchestrator auditing features to track who changed what (process modifications, job triggers).
  + Retain logs for an appropriate period based on industry regulations (e.g., HIPAA, GDPR).

#### Data Masking

* + For sensitive data (e.g., PII, financial info) typed or displayed on-screen, consider partial or full redaction in logs.
  + Leverage UiPath’s Private logging or other techniques to avoid accidental data exposure.

### Performance & Scalability

#### Process Design

* + Design automations with minimal unnecessary wait times or loops.
  + Use background automation (simulate type, simulate click) whenever possible to free up the robot’s interface.

#### Parallelization

* + When processing large data sets, leverage multiple queue items to handle them concurrently across multiple robots.
  + Ensure data dependencies are properly handled so concurrent processing does not introduce conflicts.

#### Monitoring Robot Utilization

* + Track how busy each robot is via Orchestrator dashboards.
  + Evaluate if tasks can be optimized or if new robots are needed for peak loads.

### Documentation & Knowledge Sharing

#### Inline Annotations

* + Add annotations to important or non-obvious activities to help future developers understand the logic.

#### Process Documentation

* + Keep a separate Process Definition Document (PDD) or Solution Design Document (SDD) describing the business process, assumptions, and how automation is achieved.
  + Update these documents as the process evolves.

#### Hand-Off Checklists

* + When transitioning ownership or onboarding new team members, provide a quick reference on how to deploy, run, and troubleshoot each process.
  + Include relevant links to runbooks, Orchestrator assets, queue definitions, and environment configurations.

## Appsmith (Low-Code)

### Application Structure

* + For each Appsmith application, group related pages (screens) into modules to avoid a single, monolithic design.
  + Maintain consistent naming for pages, datasources, and JS objects.

### Data Integration & APIs

* + Expose your FastAPI microservices to Appsmith through secured endpoints (OAuth2 or API keys).
  + Avoid embedding complex data transformations in Appsmith—delegate to the back-end whenever possible.

### UI/UX Consistency

* + Adhere to a common design system or style guide (colors, fonts, spacing).
  + Use global theming to ensure brand consistency.

### Security

* + Never store sensitive data (tokens, secrets) in client-side variables or text fields.
  + Validate all user inputs in FastAPI or a separate service rather than relying on client-side checks.

## React (Front-End)

### Project Structure

* + Use a well-structured folder layout (e.g., components/, pages/, hooks/, services/).
  + Adopt a predictable naming convention (PascalCase for components, camelCase for variables/functions).

### State Management

* + Choose a standardized approach for state management (e.g., React Context API, Redux, Zustand).
  + Keep domain-specific logic out of the UI components—use custom hooks or service modules.

### Code Quality

* + Follow an established style guide (e.g., Airbnb) enforced by ESLint and Prettier.
  + Avoid large components that mix multiple concerns; refactor into smaller, reusable components.

### Testing

* + Write unit tests using Jest and React Testing Library.
  + Cover critical paths: user interactions, form submissions, data fetching.

### Security & Performance

* + Use HTTPS for all communication via Nginx.
  + Implement client-side security best practices (e.g., safe handling of tokens, XSS prevention with React’s built-in protections).
  + Optimize performance (lazy-load components, code splitting, caching).

## Python FastAPI (Back-End)

FastAPI is a modern, high-performance web framework for building APIs with Python 3.7+ based on standard Python type hints. It offers an intuitive developer experience, built-in data validation via Pydantic, and asynchronous support powered by Starlette. Below are comprehensive guidelines and examples to help you set up, structure, and maintain your FastAPI services in a robust and scalable manner.

### 2.4.1 Project Layout & Structure

#### Recommended Folder Structure

* + app/: Main application package
    - main.py: Entry point of the FastAPI application
    - routers/: Contains all your route (endpoint) definitions, organized by feature/module
    - models/: Database models (e.g., SQLAlchemy models)
    - schemas/: Pydantic models for request and response validation
    - services/: Business logic (e.g., functions or classes that interact with models, external APIs)
    - core/ or utils/: Shared utilities, configurations, or helper modules
    - tests/: Testing code, possibly split into unit, integration, and end-to-end subfolders
  + requirements.txt or poetry.lock/pyproject.toml: Dependency management
  + .env (excluded from version control): Environment-specific variables
  + Dockerfile and docker-compose.yml (if containerizing)

Example structure:

**app/**

**├── main.py**

**├── routers/**

**│ ├── users.py**

**│ ├── auth.py**

**│ └── items.py**

**├── models/**

**│ ├── user.py**

**│ └── item.py**

**├── schemas/**

**│ ├── user\_schema.py**

**│ └── item\_schema.py**

**├── services/**

**│ ├── user\_service.py**

**│ └── item\_service.py**

**├── core/**

**│ ├── config.py**

**│ └── security.py**

**├── utils/**

**│ └── logger.py**

**└── tests/**

**├── test\_users.py**

**├── test\_items.py**

**└── conftest.py**

This structure keeps your code organized and aligns with typical FastAPI best practices.

#### Naming Conventions

* + Modules: snake\_case for Python files (e.g., user\_service.py, auth.py)
  + Classes: PascalCase (e.g., UserService)
  + Functions/Variables: snake\_case (e.g., create\_user(), user\_repository)
  + Pydantic Schemas: PascalCase (e.g., UserCreate, ItemUpdate)

#### Application Initialization

* + In main.py, create a FastAPI instance and include routers (see more in 2.4.2 Endpoints & Routing).
  + Load configurations (database URL, secrets, etc.) from a .env file or environment variables.

**# app/main.py**

**from fastapi import FastAPI**

**from app.routers import users, items**

**from app.core.config import settings**

**app = FastAPI(title=settings.PROJECT\_NAME, version="1.0.0")**

**# Register routers**

**app.include\_router(users.router, prefix="/users", tags=["users"])**

**app.include\_router(items.router, prefix="/items", tags=["items"])**

### Endpoints & Routing

#### Grouping by Feature

* + Group related endpoints (e.g., user-related actions) in a single file (users.py) to keep code organized.
  + Use FastAPI’s APIRouter to modularize routes.

**# app/routers/users.py**

**from fastapi import APIRouter, Depends**

**from app.schemas.user\_schema import UserCreate, UserResponse**

**from app.services.user\_service import create\_user\_service, get\_user\_service**

**router = APIRouter()**

**@router.post("/", response\_model=UserResponse)**

**def create\_user(payload: UserCreate):**

**return create\_user\_service(payload)**

**@router.get("/{user\_id}", response\_model=UserResponse)**

**def get\_user(user\_id: int):**

**return get\_user\_service(user\_id)**

#### HTTP Methods & Naming

* + Use HTTP methods appropriately: GET for retrieval, POST for creation, PUT/PATCH for updates, DELETE for removals.
  + Align route naming with resources. For example, GET /users/{id} retrieves a user by ID.

#### Versioning

* + For major changes, consider versioning your API routes (e.g., /v1/users, /v2/users).
  + Keep older versions as read-only or deprecated until fully phased out.

#### OpenAPI & Documentation

* + FastAPI automatically generates interactive docs (/docs or /redoc) from your route signatures and Pydantic models.
  + Provide descriptive docstrings and Pydantic field descriptions for clarity.

### Pydantic Models & Data Validation

#### Request/Response Models

* + Create Pydantic models (CreateUser, UpdateUser, UserResponse, etc.) to validate incoming request bodies and shape outgoing responses.
  + Validate fields (e.g., length, regex, numeric range) within these models.

**# app/schemas/user\_schema.py**

**from pydantic import BaseModel, EmailStr, Field**

**class UserCreate(BaseModel):**

**email: EmailStr**

**password: str = Field(..., min\_length=8, max\_length=64)**

**full\_name: str**

**class UserResponse(BaseModel):**

**id: int**

**email: EmailStr**

**full\_name: str**

#### Validation Tips

* + Use Field() for advanced validation.
  + Use EmailStr or custom types for stricter checks.
  + Leverage model inheritance for partial updates (e.g., UserUpdate inherits from BaseModel and sets fields as optional).

### Database Integration

Most commonly, you’ll use PostgreSQL with an ORM or query builder. Popular choices include:

* [SQLAlchemy](https://www.sqlalchemy.org/) (traditional ORM)
* [Gino](https://python-gino.org/) (async ORM)
* [Tortoise ORM](https://tortoise-orm.readthedocs.io/) (async ORM)

Example with SQLAlchemy + Alembic:

#### Database Configuration

**# app/core/config.py**

**from pydantic import BaseSettings**

**class Settings(BaseSettings):**

**PROJECT\_NAME: str = "MyFastAPIProject"**

**DATABASE\_URL: str = "postgresql+psycopg2://user:pass@localhost:5432/mydb"**

**# Load from .env, environment variables, etc.**

**class Config:**

**env\_file = ".env"**

**settings = Settings()**

#### SQLAlchemy Setup

**# app/models/database.py**

**from sqlalchemy import create\_engine**

**from sqlalchemy.ext.declarative import declarative\_base**

**from sqlalchemy.orm import sessionmaker**

**from app.core.config import settings**

**engine = create\_engine(settings.DATABASE\_URL, echo=False)**

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

**Base = declarative\_base()**

#### Database Models

**# app/models/user.py**

**from sqlalchemy import Column, Integer, String**

**from app.models.database import Base**

**class User(Base):**

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

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

**email = Column(String, unique=True, index=True, nullable=False)**

**hashed\_password = Column(String, nullable=False)**

**full\_name = Column(String, nullable=True)**

#### Alembic for Migrations

* + Use Alembic to handle schema changes.
  + Configure alembic.ini and env.py to point to your DATABASE\_URL and Base.metadata.
  + Example commands:

**alembic revision --autogenerate -m "create user table"**

**alembic upgrade head**

#### Dependency Injection

**# app/routers/dependencies.py**

**from typing import Generator**

**from app.models.database import SessionLocal**

**def get\_db() -> Generator:**

**db = SessionLocal()**

**try:**

**yield db**

**finally:**

**db.close()**

**# Usage in an endpoint**

**@router.get("/{user\_id}")**

**def get\_user(user\_id: int, db: Session = Depends(get\_db)):**

**...**

### Business Logic & Services

#### Service Layer

* + Keep your core logic in dedicated service files or classes (e.g., UserService).
  + The router (endpoint) calls into the service, which interacts with the database or other external systems.

**# app/services/user\_service.py**

**from sqlalchemy.orm import Session**

**from app.schemas.user\_schema import UserCreate, UserResponse**

**from app.models.user import User**

**from app.utils.security import get\_password\_hash**

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

**user = User(**

**email=user\_data.email,**

**hashed\_password=get\_password\_hash(user\_data.password),**

**full\_name=user\_data.full\_name**

**)**

**db.add(user)**

**db.commit()**

**db.refresh(user)**

**return UserResponse.from\_orm(user)**

#### Why a Service Layer?

* + Decouples business logic from presentation logic (endpoints).
  + Eases unit testing and code reuse.

### Authentication & Authorization

FastAPI provides built-in support for OAuth2 flows. You can also integrate libraries like:

* fastapi-users – user management & authentication flows
* Authlib – OAuth and JWT solutions
* [fastapi-jwt-auth](https://github.com/IndominusByte/fastapi-jwt-auth) – JWT authentication

Example of OAuth2 with JWT:

**# app/core/security.py**

**from datetime import datetime, timedelta**

**from jose import JWTError, jwt**

**from fastapi import Depends, HTTPException, status**

**from fastapi.security import OAuth2PasswordBearer**

**SECRET\_KEY = "replace\_with\_real\_secret"**

**ALGORITHM = "HS256"**

**ACCESS\_TOKEN\_EXPIRE\_MINUTES = 60**

**oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="auth/login")**

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

**to\_encode = data.copy()**

**expire = datetime.utcnow() + timedelta(minutes=expires\_delta)**

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

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

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

**try:**

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

**user\_id: str = payload.get("sub")**

**if user\_id is None:**

**raise credentials\_exception**

**except JWTError:**

**raise credentials\_exception**

**# fetch user from DB or cache**

**...**

#### Password Hashing

* + Use a strong hashing library like [Passlib](https://passlib.readthedocs.io/en/stable/) or bcrypt to store hashed passwords securely.

#### Role-Based Access Control (RBAC) or Attribute-Based Access Control (ABAC)

* + Define roles and check permissions at the router or service level.
  + For more complex scenarios, consider an external policy engine (e.g., Oso).

### Error Handling & Exception Management

#### Built-In HTTPException

* + Raise HTTPException with a status code and detail message for expected errors (e.g., resource not found).
  + Wrap unexpected failures in try/except blocks and convert them to a user-friendly error code.

**from fastapi import HTTPException**

**if not user:**

**raise HTTPException(status\_code=404, detail="User not found")**

#### Custom Exception Handlers

* + Register exception handlers for specific errors (e.g., database errors, validation errors) for consistent error responses.

**from fastapi import FastAPI, Request**

**from sqlalchemy.exc import IntegrityError**

**app = FastAPI()**

**@app.exception\_handler(IntegrityError)**

**async def integrity\_error\_handler(request: Request, exc: IntegrityError):**

**return JSONResponse(**

**status\_code=400,**

**content={"detail": "Database integrity error occurred."},**

**)**

#### Logging Errors

* + Use Python’s built-in logging module

### Testing

Use a combination of Pytest, HTTPX (for calling the FastAPI server), and pytest-asyncio (if you’re using async database calls).

#### Example Test Structure

# app/tests/test\_users.py

import pytest

from fastapi.testclient import TestClient

from app.main import app

client = TestClient(app)

def test\_create\_user():

response = client.post("/users/", json={

"email": "test@example.com",

"password": "somepassword",

"full\_name": "Test User"

})

assert response.status\_code == 200

data = response.json()

assert data["email"] == "test@example.com"

#### Database Setup for Tests

* + Use a separate test database to avoid polluting production or dev data.
  + Implement fixtures to create and tear down test data automatically.

#### Coverage & CI

* + Integrate tests into a CI pipeline (e.g., GitHub Actions, GitLab CI) to ensure every commit is tested.
  + Aim for high coverage on critical paths (authentication, core logic).

### Security & Performance

1. HTTPS & Nginx
   * Terminate SSL at Nginx. Connect Nginx to FastAPI over a secure network channel.
   * Enforce secure headers (e.g., Strict-Transport-Security, X-Frame-Options).
2. Cross-Origin Resource Sharing (CORS)
   * If your front end (React or Appsmith) is on a different domain, configure fastapi.middleware.cors.CORSMiddleware properly:

from fastapi.middleware.cors import CORSMiddleware

origins = ["https://example.com", "http://localhost:3000"]

app.add\_middleware(

CORSMiddleware,

allow\_origins=origins,

allow\_credentials=True,

allow\_methods=["\*"],

allow\_headers=["\*"],

)

#### Rate Limiting

* + Consider using [slowapi](https://pypi.org/project/slowapi/) or [fastapi-limiter](https://pypi.org/project/fastapi-limiter/) for rate-limiting if you expect abusive traffic patterns.

#### Asynchronous Performance

* + Use async/await for I/O-bound operations (database calls, HTTP requests).
  + Avoid blocking calls in the main event loop. If you have CPU-intensive tasks, consider background workers (e.g., Celery or RQ) or Python’s multiprocessing.

#### Caching

* + For frequently accessed data that doesn’t change often, consider an external cache (e.g., Redis).
  + Libraries like [fastapi-cache](https://pypi.org/project/fastapi-cache2/) can simplify caching at the route or function level.

### Logging & Monitoring

#### Logging Configuration

* + Configure a consistent logging setup with logging.config.dictConfig or a similar approach.
  + Use structured logging (JSON) for easier parsing in centralized log management tools (e.g., ELK, Datadog).

#### Metrics & Tracing

* + Integrate [Prometheus](https://pypi.org/project/prometheus-fastapi-instrumentator/) for metrics like request counts, latencies, etc.
  + Use a distributed tracing solution like [OpenTelemetry](https://opentelemetry.io/) with FastAPI for end-to-end request tracing.

#### Health Checks

* + Create a simple /health or /ping endpoint to quickly check if your service is operational.
  + Optionally, add deeper checks (e.g., DB connectivity, cache availability) for more comprehensive health status.

### Deployment & CI/CD

#### Containerization

* + Create a minimal Dockerfile that includes only runtime dependencies. Example:

FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

* + Keep build context small and avoid copying unnecessary files (e.g., tests if not needed at runtime).

#### Orchestration

* + Use Kubernetes, Docker Swarm, or AWS ECS/Fargate to run containers.
  + Configure environment variables or secrets (DB credentials, JWT secrets) via your orchestration platform’s secret management.

#### CI/CD Pipeline

* + Use tools like GitHub Actions, GitLab CI, or Jenkins to automate building, testing, and pushing Docker images.
  + Incorporate automated checks: code linting, unit tests, security scans (e.g., [Bandit](https://pypi.org/project/bandit/) for Python, container vulnerability scans).

#### Blue-Green / Rolling Deployments

* + Reduce downtime and risk by using a rolling update strategy or a blue-green deployment approach.
  + Validate new versions in a staging environment or traffic-splitting scenario before fully switching production traffic.

### Advanced Topics & Recommendations

#### GraphQL Support

* + If GraphQL is required, explore [Strawberry](https://strawberry.rocks/) or [Ariadne](https://ariadnegraphql.org/) with FastAPI integration.

#### WebSockets & Real-Time Features

* + FastAPI natively supports WebSockets for real-time communication.
  + Consider using [Socket.IO](https://pypi.org/project/python-socketio/) or [FastAPI-SocketIO](https://github.com/pyropy/fastapi-socketio) for more advanced real-time patterns.

#### Tasks & Async Workers

* + Offload CPU-intensive or long-running tasks to Celery or RQ workers.
  + Use message brokers like Redis or RabbitMQ to queue tasks.

#### Multi-Tenancy

* + For multi-tenant architectures, consider separate DB schemas or a carefully designed data model with a tenant\_id field.
  + Use a middleware or dependency to apply tenant context for each request.

#### Security Hardening

* + Regularly apply security patches for Python packages.
  + Run [dependabot](https://github.com/dependabot) or similar to keep dependencies updated.

## PostgreSQL / Database (Updated)

### Context & SQL Engine

* + Unless otherwise specified by the client, **PostgreSQL** is the **primary database engine**.
  + In specific cases (e.g., lightweight prototyping or local RPA scenarios), **SQLite** may be considered.
  + Non-relational (NoSQL) or other SQL engines (MySQL, SQL Server, Oracle) are adopted only if explicitly requested by the client or if there are compelling technical reasons to do so.
  + Tables dedicated to low-coding products (budibase and appsmith) can deviate from our standards. Also, tables used just for loading external data can maintain the external data conventions.

### Recommended Software

* + **DBeaver** is recommended as a unified database client to manage multiple types of databases (PostgreSQL, SQLite, etc.).
  + **Migration/versioning tools** (e.g., **Alembic** in Python) are strongly advised to maintain a history of schema changes and facilitate version control—especially for Python FastAPI projects.

### Number of Databases & Naming

* + **One database per client**: each client has a dedicated database (named after the client) to simplify potential porting or handover.
  + If multiple environments are needed (dev, staging, production), add prefixes or suffixes (e.g., <client>\_dev, <client>\_stg, <client>\_prod).
  + In multi-tenant architectures, isolation can be achieved through separate databases or schemas, depending on security and scalability requirements.

### Architectural Rules

* + **Normalization**: tables are designed according to normal forms, minimizing redundancy and ensuring referential integrity (foreign keys). <https://learn.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description>
  + **Table Naming**: each table follows the prefix\_tablename structure in lowercase. Common prefixes include:
    - **D\_** (dictionary), **T\_** (transactional), **L\_** (log), **R\_** (relationship), **T\_R\_** (transactional relationship), **Q\_** (queue), **Z\_** (transcoding), **S\_** (setup), **A\_** (appendix), **B\_** (Budibase), **scratch\_** (temporary staging).
  + **Coded Columns**: if there is a column named cod\_stato\_nascita in t\_persona, the corresponding decoding table is d\_stato\_nascita. Transactional or log tables use id as primary keys, while dictionary tables use Code.

### Schema Design & Versioning

* + **Migrations**: tools like Alembic (Python) or Flyway help manage updates, rollbacks, and version control of DB schema changes.
  + **Git Repository**: all important SQL scripts (DDL/DML) and migration files must be tracked in the project’s Git repo. If no dedicated repo exists, create one specifically for DB versioning.

### SQL Query Rules

* + **Aliases in SELECT**: use meaningful aliases to clarify column output.
  + **Aliases in FROM**: generally avoid them to reduce ambiguity, unless referencing the same table multiple times with different joins.
  + **Formatting**: maintain consistency (uppercase for SQL keywords like SELECT, FROM, WHERE; lowercase for table/column names).
  + **Complex Queries**: for very complex logic, consider SQL functions or stored procedures. In RPA contexts (e.g., UiPath), avoid dynamically building raw SQL strings to reduce the risk of injection.

### Indexing & Performance Tuning

* + Create indexes on columns frequently used in WHERE clauses or joins.
  + Monitor performance with EXPLAIN/EXPLAIN ANALYZE.
  + For large transactional data volumes, consider table partitioning or separate log tables (e.g., L\_).

### Backup & Logging

* + **Daily Backups**: all databases must be backed up at least once a day (full backup). The developer is responsible for verifying the initial setup, and the control room oversees daily operations.
  + **T\_ Tables**: since transactional tables can undergo updates/deletes, consider creating a corresponding L\_ (log) table with triggers that record each event (cod\_evento = insert/update/delete) and data\_aggiornamento, enabling recovery from malicious or accidental changes.
  + **PITR** (Point-in-Time Recovery): in production, consider using WAL backups and streaming replication (PostgreSQL) for recovery to any specific time.
  + **Restore Testing**: periodically run restore tests to ensure backups are valid and fully restorable.

## Blob / S3 File Server

### File Handling

* + Store large or binary files in S3/Blob storage rather than Postgres.
  + Maintain a clear folder/key structure that can be easily navigated and managed.

### Access Control

* + Use service accounts or IAM roles to control read/write access.
  + Presign URLs for temporary access (if needed in Appsmith or React) instead of distributing permanent URLs.

### Lifecycle Policies

* + Implement lifecycle rules (e.g., archival or deletion) for data retention requirements and cost optimization.

# Deployment & Infrastructure

## Containerization

* + Use Docker for packaging (FastAPI, React if SSR or using a Node-based build).
  + Maintain minimal base images and handle secrets securely (avoid hardcoding in Dockerfiles).

## Nginx Configuration

* + Set up Nginx as a reverse proxy in front of Appsmith and custom applications.
  + Enforce HTTPS (TLS certificates) and use secure ciphers.
  + Use caching headers, GZIP compression, and rate limiting if appropriate.

## CI/CD Pipeline

* + Set up automated build/test steps on every commit (e.g., GitHub Actions, GitLab CI).
  + Automate deployment to dev/staging/production environments with approvals and version tagging.

## Infrastructure as Code

* + Use Terraform or similar for provisioning cloud resources (S3, Postgres, etc.).
  + Keep environment configurations (dev, staging, prod) consistent and well-documented.

## Environment Configuration

* + Store secrets and credentials in a secure environment (vault, environment variables, or K8s secrets).
  + Distinguish between config variables (non-secret) and secrets clearly.
  + Include a .env.example in each project to document required configurations.

# Quality Assurance & Testing

## Test Strategy

* + **Unit Tests**: For all custom code (Python, JavaScript).
  + **Integration Tests**: For API endpoints and critical workflows (e.g., combined RPA + back-end calls).
  + **UI/End-to-End Tests**: For Appsmith & React front-end flows using frameworks like Cypress, Playwright, or Selenium.

## Code Reviews

* + Enforce mandatory code reviews for all merges to main or master.
  + Review both logic and style—catch architectural, security, and performance issues early.

## Automated Testing in CI

* + Run unit and integration tests automatically on pull requests.
  + Fail builds that do not meet minimum coverage thresholds.

## Performance Testing

* + Test with realistic data volumes and concurrency for critical flows.
  + Identify bottlenecks (database queries, external APIs, or code loops).

## Security Testing

* + Periodically run vulnerability scans (SAST, DAST, container scans).
  + Perform penetration testing or use third-party auditors for compliance-driven industries.

# 5. Monitoring & Logging

## Logging

* + Standardize log formats (JSON or text with fields).
  + Log at appropriate levels (DEBUG, INFO, WARN, ERROR) and centralize logs (e.g., ELK stack, Datadog, Splunk).

## Metrics & Observability

* + Capture metrics (CPU, memory, I/O, latency) and instrument custom metrics (e.g., requests/s, error rates) using Prometheus/Grafana or another solution.
  + Set up alerts for threshold breaches (e.g., 5xx errors, high memory usage).

## Tracing

* + Implement distributed tracing (OpenTelemetry, Jaeger) especially if multiple microservices are involved.
  + Correlate logs, metrics, and traces to troubleshoot issues quickly.

## Incident Response

* + Define an on-call rotation or escalation policy.
  + Document procedures for investigating, communicating, and resolving incidents.

# Governance & Compliance

## Policy Adherence

* + Document standards in a living repository (e.g., Confluence, Notion, or Git-backed markdown).
  + Conduct periodic reviews to update for new technologies or organizational changes.

## Roles & Responsibilitie

* + Clearly define who owns each component (UiPath, Appsmith, back-end, front-end, DB).
  + Establish a security champion or governance role to ensure best practices are followed.

## Regulatory Compliance

* + Consider data privacy laws (GDPR, CCPA) if handling user data.
  + Comply with any industry-specific regulations (HIPAA, PCI-DSS) through encryption, access controls, and audit logging.

## Change Management

* + Maintain a change log or release notes for major upgrades or new features.
  + Use a ticketing system (Jira, Trello, Asana) to track requests, bugs, and improvements.

# Documentation & Knowledge Sharing

## Project README

* + Each repository (UiPath, Appsmith config, React front-end, FastAPI back-end) should have a README outlining setup, run commands, and key architectural decisions.

## API Documentation

* + FastAPI automatically generates API docs—ensure that they are kept accurate with docstrings and Pydantic models.

## User Guides & Tutorial

* + For each RPA workflow or Appsmith module, have a short how-to or wiki page so new team members can get up to speed.

## Onboarding & Training

* + Provide training sessions or internal demos regularly to keep the team up to date on best practices and new features.