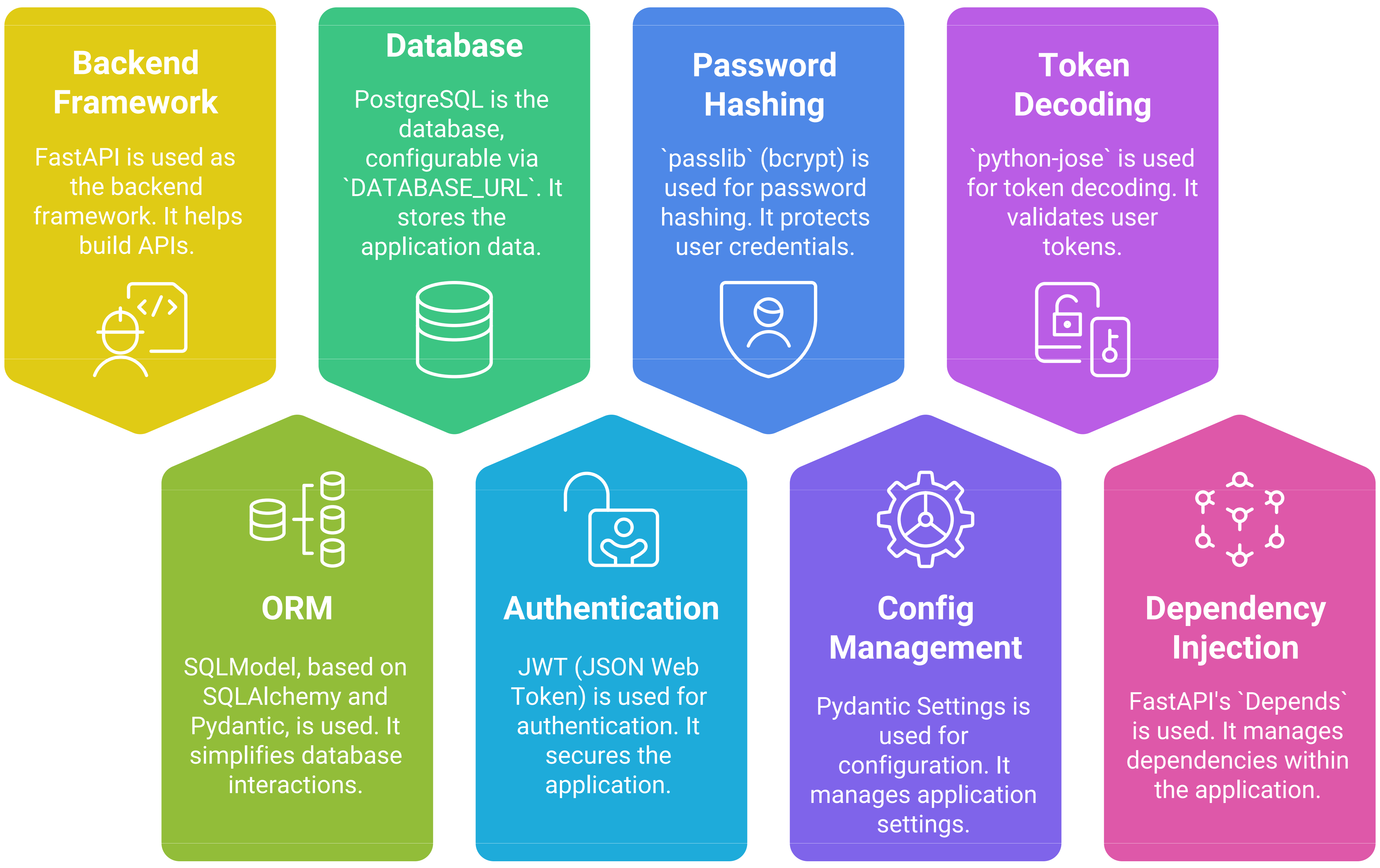
# Project Summary: JWT Auth + RBAC API using FastAPI and SQLModel

Tech Stack

- **Backend Framework**: FastAPI
- **ORM**: SQLModel (based on SQLAlchemy + Pydantic)
- **Database**: PostgreSQL (configurable via **DATABASE_URL**)
- **Authentication**: JWT (JSON Web Token)
- **Password Hashing**: **passlib** (bcrypt)
- **Config Management**: Pydantic Settings
- **Token Decoding**: **python-jose**
- **Dependency Injection**: FastAPI's **Depends**

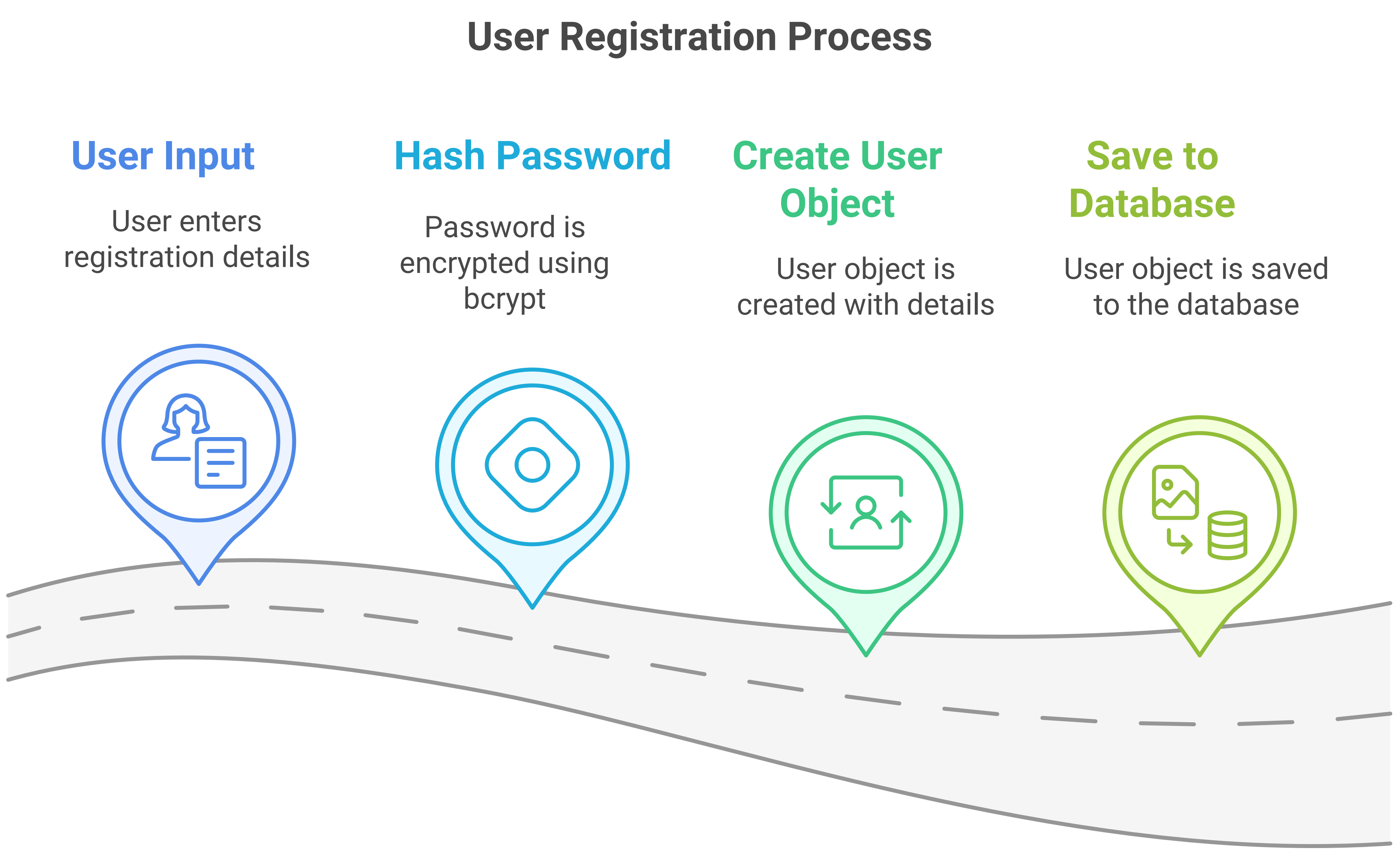
Project Dependencies



Backend Framework
FastAPI is used as the backend framework. It helps build APIs.
Database
PostgreSQL is the database, configurable via `DATABASE_URL`. It stores the application data.
Password Hashing
`passlib` (bcrypt) is used for password hashing. It protects user credentials.
Token Decoding
`python-jose` is used for token decoding. It validates user tokens.
ORM
SQLModel, based on SQLAlchemy and Pydantic, is used. It simplifies database interactions.
Authentication
JWT (JSON Web Token) is used for authentication. It secures the application.
Config Management
Pydantic Settings is used for configuration. It manages application settings.
Dependency Injection
FastAPI's `Depends` is used. It manages dependencies within the application.

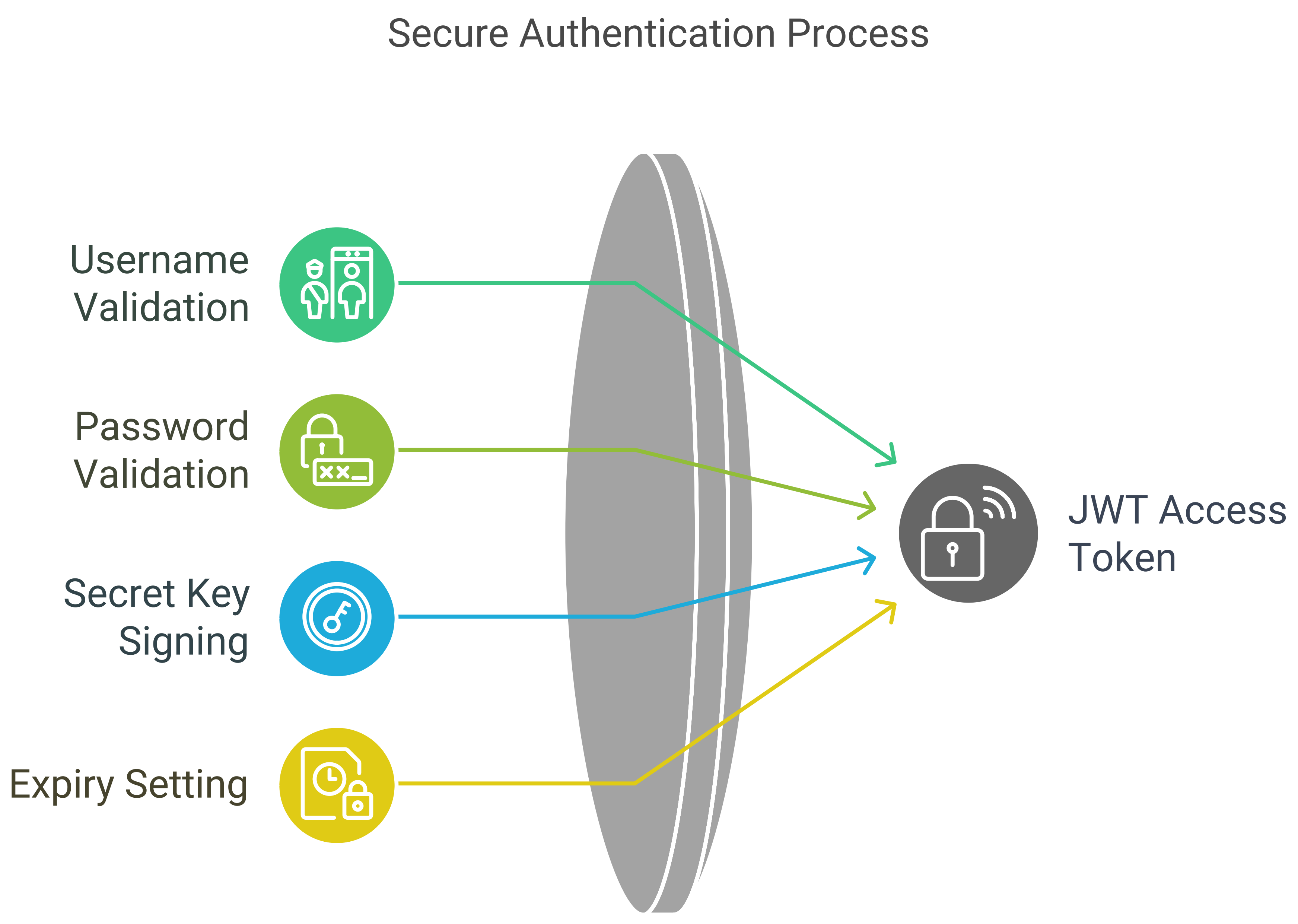
## Core Features

1. User Registration (/register)
   - Creates a new user with:
     - **username**
     - **password** (hashed with bcrypt)
     - **role** (e.g., "admin", "user")
   - Saves the user to the database using SQLModel.


User Registration Process
User Input
User enters registration details
Hash Password
Password is encrypted using bcrypt
Create User Object
User object is created with details
Save to Database
User object is saved to the database

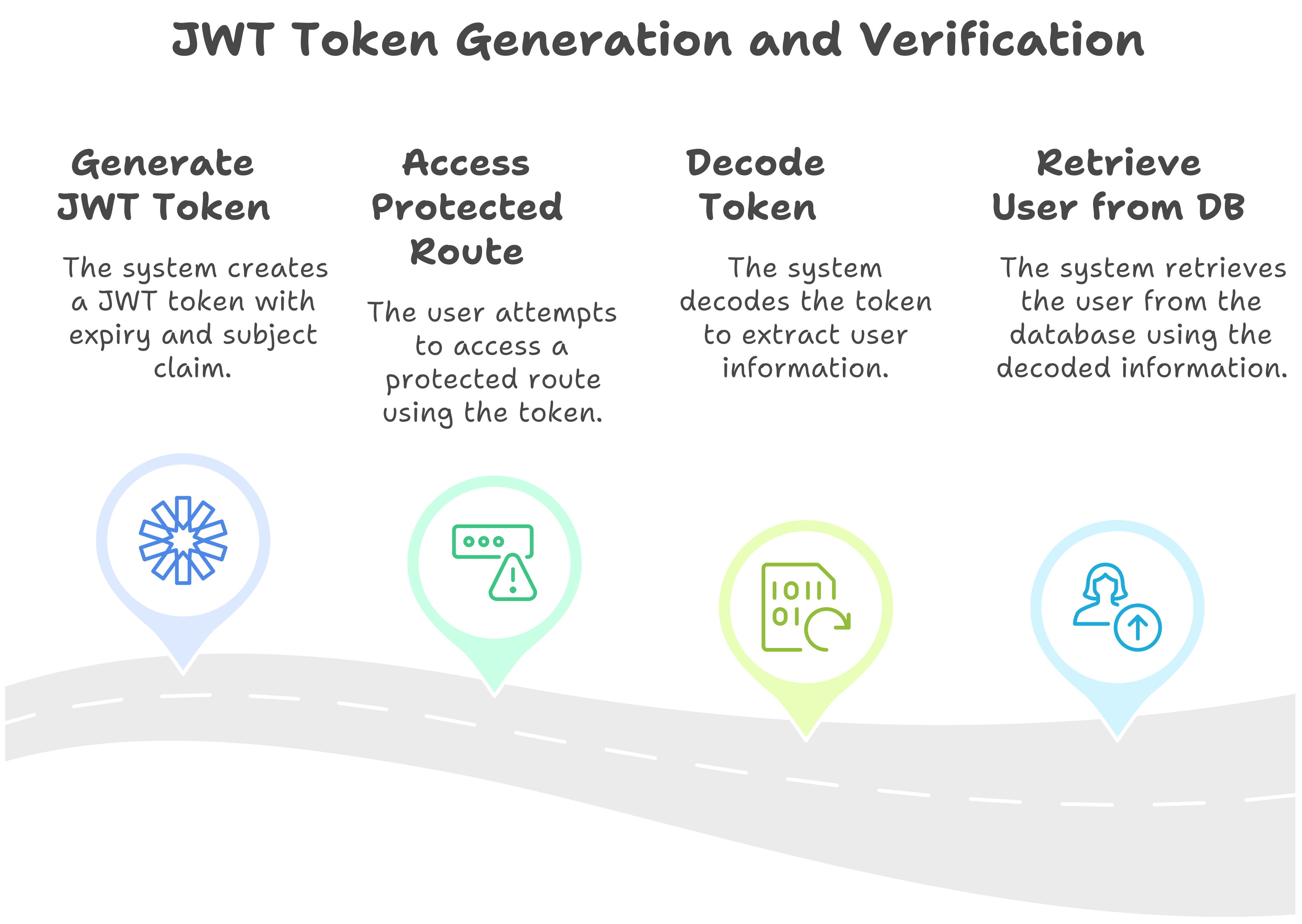
2. Login (/login)

- Validates the username and password.
- If successful, returns a **JWT access token** signed with a secret key and expiry.



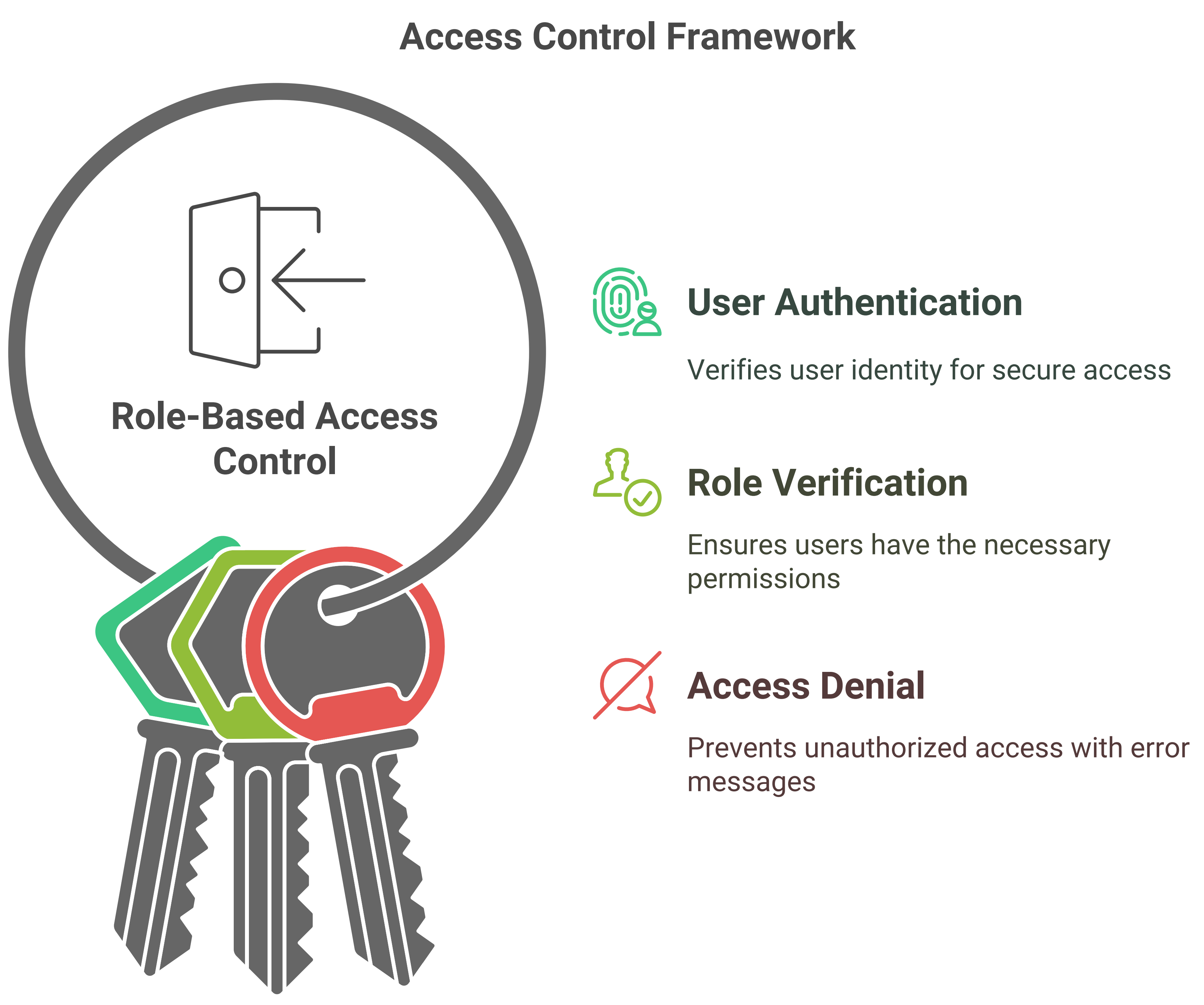
3. JWT Token Handling

- Uses the **create_access_token** function to generate tokens with expiry and **sub** (subject) claim set to the username.
- Token is required for accessing protected routes.
- The token is decoded in **get_current_user** to retrieve the user from DB.

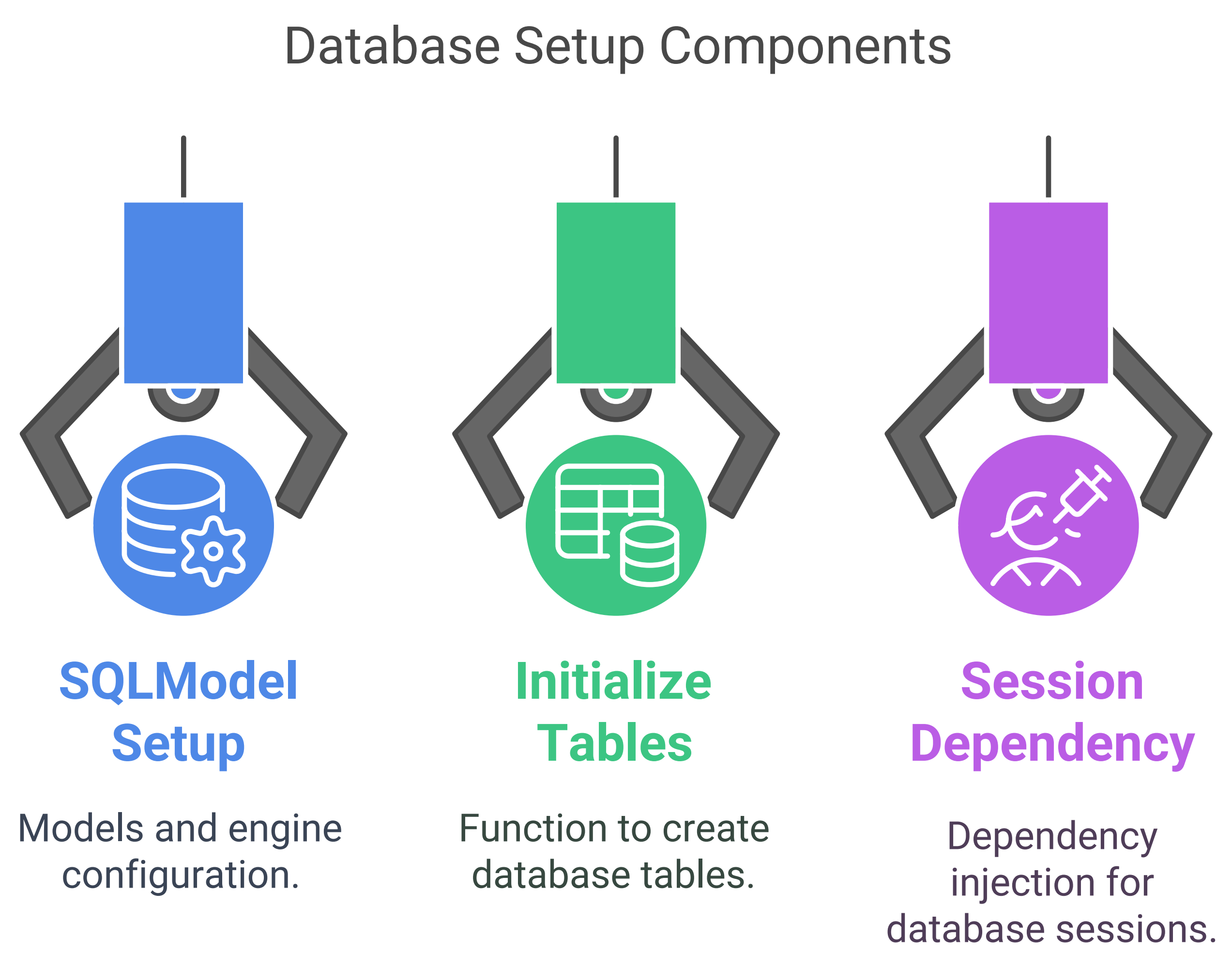

JWT Token Generation and Verification
Generate JWT Token
The system creates a JWT token with expiry and subject claim.
Access Protected Route
The user attempts to access a protected route using the token.
Decode Token
The system decodes the token to extract user information.
Retrieve User from DB
The system retrieves the user from the database using the decoded information.
1011
01

4. Role-Based Access Control (RBAC)

- **require_role(role: str)** is a reusable dependency that:
  - Verifies the authenticated user has the required role.
  - Raises **403 Forbidden** if access is denied.
- This is useful for endpoints like:
- **@router.get("/admin-dashboard")**
- **def admin_only(user: User = Depends(require_role("admin"))):**
- **return {"message": "Welcome admin"}**


Access Control Framework
Role-Based Access Control
User Authentication
Verifies user identity for secure access
Role Verification
Ensures users have the necessary permissions
Access Denial
Prevents unauthorized access with error messages

5. Database Setup
- SQLModel-based models and database engine setup (**engine**)
- **create_db_and_tables()** to initialize tables
- **get_session()** dependency for session injection in routes/services


Database Setup Components
SQLModel Setup
Models and engine configuration.
Initialize Tables
Function to create database tables.
Session Dependency
Dependency injection for database sessions.

6. Security Utilities

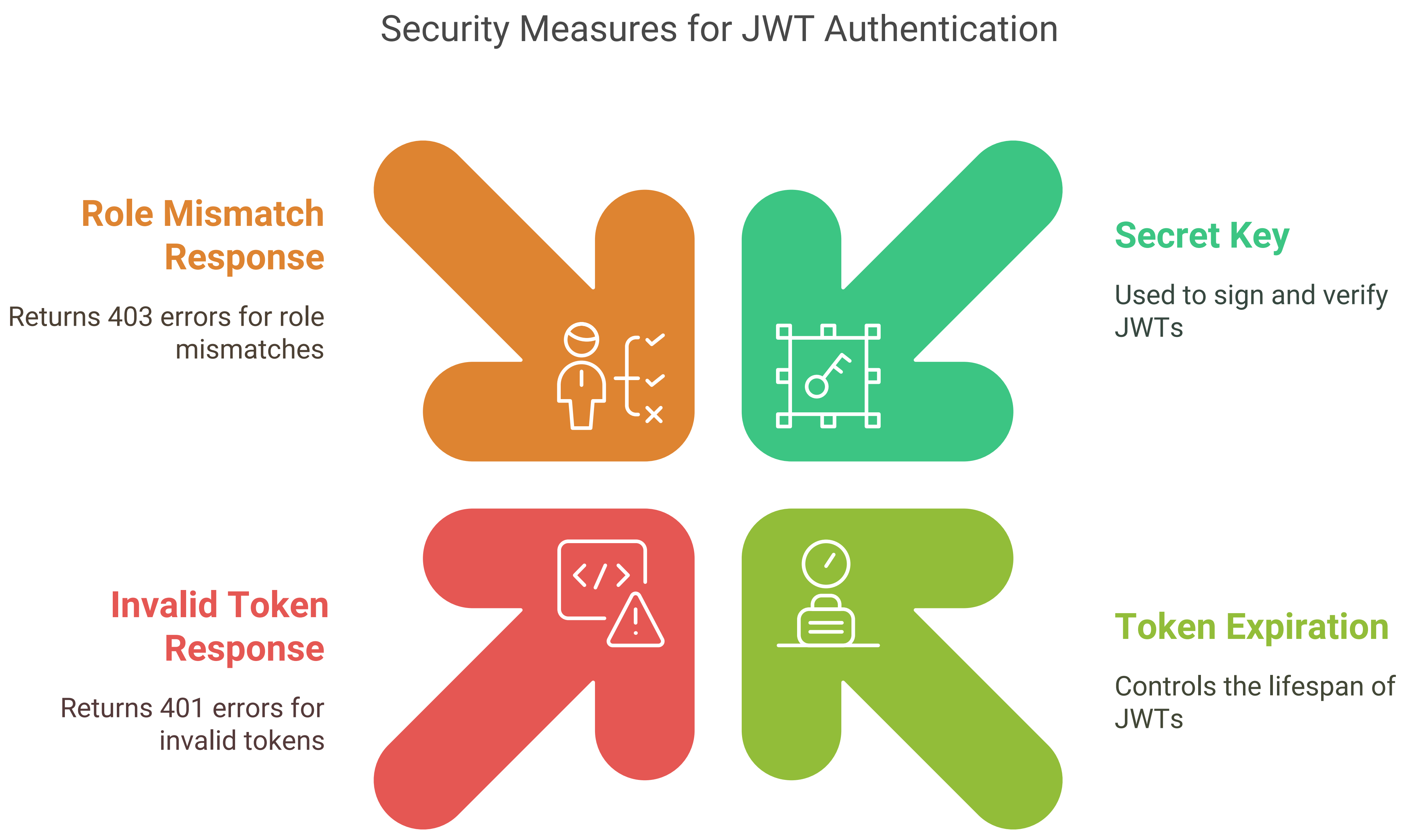
- Password hashing and verification (**get_password_hash**, **verify_password**)
- JWT creation and decoding using **python-jose**

Security functions
JWT processing
Creation and decoding of JSON web tokens.
Password handling
Hashing and verification of passwords.

## Security Highlights

- Tokens are signed with **SECRET_KEY** using the algorithm defined in **.env** (**HS256**, etc.)

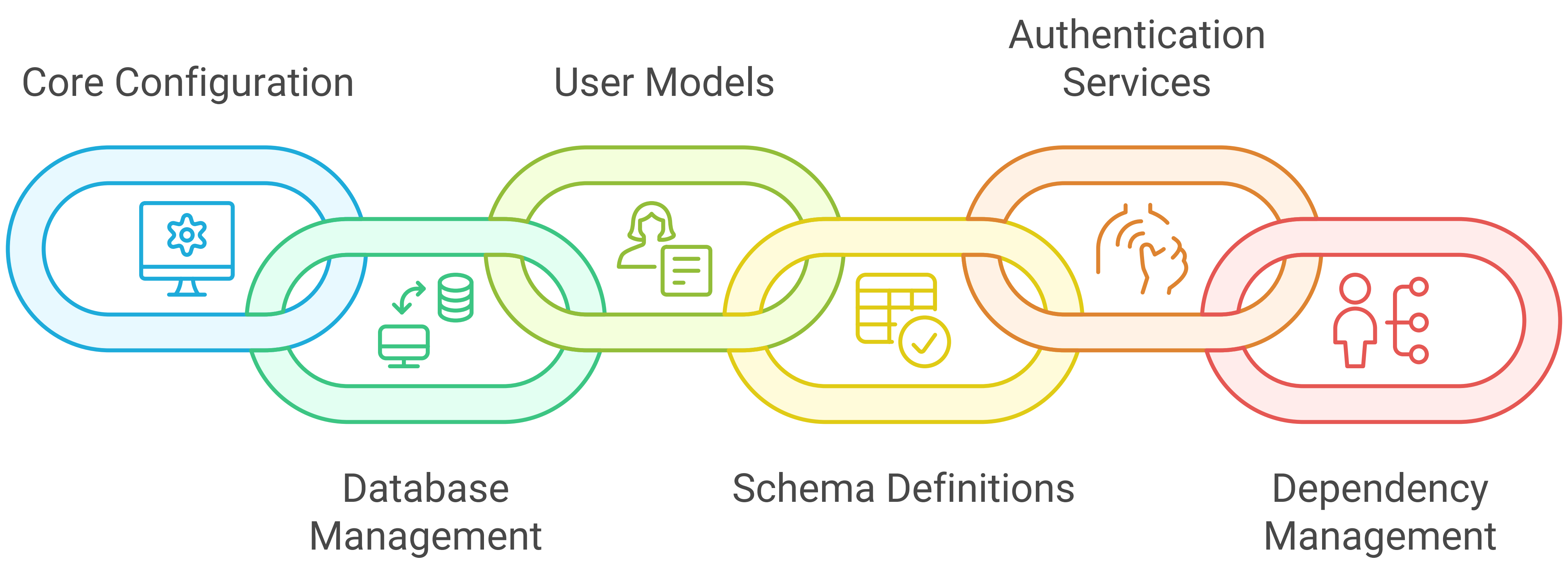
- Token expiration is controlled via **ACCESS_TOKEN_EXPIRE_MINUTES**
- Invalid or expired tokens return appropriate **401** responses
- Role mismatches return **403** errors


Security Measures for JWT Authentication
Role Mismatch Response
Returns 403 errors for role mismatches
Secret Key
Used to sign and verify JWTs
Invalid Token Response
Returns 401 errors for invalid tokens
Token Expiration
Controls the lifespan of JWTs

## Code Structure

```
app/  ├─── core/  │
   ├─── config.py       # Loads environment config
   │    └─── security.py      # Handles password hashing and JWT logic
   ├─── db/
   │    └─── database.py      # DB engine, session, and table creation
   ├─── models/
   │    └─── user.py        # SQLModel user model
   ├─── schemas/
   │    └─── user_schema.py   # Pydantic schemas for request/response
   ├─── services/
   │    └─── auth_service.py  # Business logic for login, register
   ├─── dependencies/
   │    └─── auth.py        # get_current_user and require_role
```

Core Components of JWT Auth API



## Typical Flow

1. User registers → **hashed_password** is stored in DB.
2. User logs in → token is issued.
3. Authenticated requests → token is passed in **Authorization: Bearer <token>**.
4. FastAPI uses dependency injection to validate token and role before processing route logic.

## JWT Authentication and Authorization Flow

**User Registers**

User creates an account

**Store Hashed Password**

Password is encrypted and saved

**User Logs In**

User enters credentials to log in

**Issue Token**

System generates a JWT token

**Authenticated Request**

User sends a request with the token

**Validate Token**

FastAPI checks the token's validity

**Check Role**

FastAPI verifies user's role

**Process Route Logic**

FastAPI executes the route logic