Name

full-stack-fastapi-template

Project overview

FastAPI Project - Backend

This project is a backend developed using FastAPI. It requires Docker and Poetry for Python package and environment management.

Key Features

- Frontend built with Docker, with routes handled based on the path.
- Backend, JSON based web API based on OpenAPI.
- Automatic interactive documentation with Swagger UI.
- Adminer for database web administration.
- Traefik UI to monitor how routes are handled by the proxy.

Local Development

To start the stack, use Docker Compose with the command `docker compose up -d`. The first time you start your stack, it might take a minute for it to be ready. You can check the logs to monitor it using `docker compose logs`.

Backend Local Development

Dependencies are managed with Poetry. You can install all the dependencies from `./backend/` using `poetry install` and start a shell session with the new environment using `poetry shell`.

You can modify or add SQLModel models for data and SQL tables in `./backend/app/models.py`, API endpoints in `./backend/app/api/`, CRUD utils in `./backend/app/crud.py`.

VS Code configurations are in place to run the backend through the VS Code debugger and run the tests through the VS Code Python tests tab.

Docker Compose Override

During development, you can change Docker Compose settings that will only affect the local development environment in the file `docker-compose.override.yml`.

Backend Tests

To test the backend, run `bash ./scripts/test.sh`. The tests run with Pytest, modify and add tests to `./backend/app/tests/`.

Migrations

After changing a model, create a revision using Alembic and run the migration in the database. This will

update the tables in your database. If you don't want to use migrations, you can modify the files in `./backend/app/core/db.py` and `prestart.sh`.

Contribution

The README does not provide specific contribution guidelines. However, given the detailed instructions for local development and testing, contributors should be able to set up the project locally and make contributions as needed.

Business use cases

users.py

Route: /
Functions:

- `get_user_by_email`
- `create_user`
- `generate_new_account_email`
- `send_email`
- `get_password_hash`

Case:
Business Case:

The business case revolves around the development and implementation of a secure user management system for a Python-based application. The system will be designed to handle user registration, authentication, and email notifications.

1. User Registration: The system will allow new users to register by providing their email and password. The 'create_user' function will be used to create a new user in the database. It will first check if a user with the same email already exists. If so, it will raise an exception. Otherwise, it will create a new user and store it in the database.

2. Password Hashing: To ensure the security of user data, the system will not store passwords in plain text. Instead, it will use the 'get_password_hash' function to hash the passwords before storing them. This function uses the 'pwd_context' module to hash the password using industry-standard algorithms.

3. User Authentication: The system will allow users to authenticate using their email and password. The 'get_user_by_email' function will be used to retrieve a user from the database based on their email.

4. Email Notifications: The system will also send email notifications to users. For instance, when a new user is created, the system will send a confirmation email to the user's registered email address. The 'send_email' function will be used to send these emails. It will use the SMTP protocol and the settings provided in the 'settings' module.

By implementing this user management system, the business can ensure the secure handling of user data, improve user experience through email notifications, and streamline the process of user registration and authentication. This will ultimately lead to increased user satisfaction and retention, thereby contributing to the overall success of the business.
Route: /me
Functions: No functions

Route: /signup
Functions:

- `get_user_by_email`
- `create_user`
- `get_password_hash`

Case:
Business Case:

The business case revolves around the development and implementation of a secure user management system for a web-based application. The system is designed to handle user registration, authentication, and password management using Python and SQLAlchemy, a SQL toolkit and Object-Relational Mapping (ORM) system.

1. User Registration: The `create_user` function allows for the creation of new users in the system. It checks if a user with the same email already exists to prevent duplicate accounts. If the email is unique, the function creates a new user and, if email notifications are enabled, sends a confirmation email to the new user.

2. User Retrieval: The `get_user_by_email` function retrieves a user from the database using their email address. This function is essential for various operations like user authentication, profile management, and user-specific operations.

3. Password Management: The `get_password_hash` function is used for secure password management. It takes a plaintext password and returns a hashed version of it using the `pwd_context` module. This hashed password is then stored in the database, providing a secure way to handle user passwords.

The implementation of these functions will ensure a secure and efficient user management system. It will not only improve the user experience by providing a smooth registration and login process but also enhance the security of user data by securely handling passwords. This system can be a part of a larger application like an e-commerce platform, a social media site, or any web-based application requiring user management.
Route: /{user_id}
Functions: No functions

utils.py

Route: /test-email/
Functions:

- `generate_test_email`
- `send_email`

Case:
Business Case:

Title: Streamlining Email Communication in Python Applications

Background:
The business currently uses Python for various applications and processes. One of the key functionalities required across these applications is the ability to send emails. This could be for various purposes such as notifications, alerts, updates, or communication with users. Currently, the process of sending emails involves writing and maintaining multiple lines of code across different parts of the application. This leads to redundancy, increased chances of errors, and difficulty in managing and updating the code.

Proposed Solution:
The proposed solution is to create two Python functions, `generate_test_email` and `send_email`,

that will handle all the email communication requirements.

The `generate_test_email` function will take an email address as input and return an `EmailData` object. This function will generate an email with a test message and send it to a specified email address.

The `send_email` function will allow sending an email with a customizable subject and body. It will check for provided configurations for email variables using the `settings` module. If not, it will raise an assertion error. It will then create a new message object with the specified subject and HTML content. The sender name and email address will be set from the configurations in the `settings` module.

Benefits:
1. Code Efficiency: By centralizing the email sending functionality, we can avoid code redundancy and make the codebase cleaner and more manageable.
2. Error Reduction: Centralizing the email sending functionality will reduce the chances of errors as there will be a single point of control.
3. Flexibility: The functions allow for customizable emails, providing flexibility to the applications using them.
4. Scalability: As the business grows and the number of emails sent increases, having a dedicated function for this will make it easier to scale up the operations.

Next Steps:
1. Develop the `generate_test_email` and `send_email` functions.
2. Test the functions thoroughly to ensure they work as expected.
3. Gradually implement the functions across different applications.
4. Monitor and optimize the functions as necessary.

login.py

Route: /login/access-token
Functions:

- `authenticate`
- `create_access_token`
- `get_user_by_email`
- `verify_password`

Case:
Business Case:

The business case revolves around the development and implementation of a secure user authentication system for a web application. The system is designed to streamline the process of user authentication, ensuring that only authorized users can access the application.

The system is built using Python and leverages several functions to authenticate users based on their email and password. The functions include `authenticate`, `create_access_token`, `get_user_by_email`, and `verify_password`.

The `authenticate` function retrieves the user with the given email from the database and checks if their password is correct. If either check fails, the function returns `None`, indicating an unsuccessful login attempt. Otherwise, it returns the `User` object.

The `create_access_token` function generates a JSON Web Token (JWT) using the PyJWT library. The JWT is encoded with a secret key and includes the subject and expiration time in its payload.

The `get_user_by_email` function retrieves a user with the specified email address from the database. It returns either the retrieved user or `None` if no user was found.

The `verify_password` function checks whether the plain password matches the hashed password. If the passwords match, the function returns `True`, indicating that the user has entered the correct password; otherwise, it returns `False`.

The implementation of this system will enhance the security of the web application by ensuring that only authorized users can access it. It will also improve the user experience by providing a seamless login process. The system can be further enhanced by incorporating features such as password reset and two-factor authentication.

The cost of implementing this system will include the development and testing time, as well as any necessary software or hardware upgrades. However, the benefits of improved security and user experience are expected to outweigh these costs.

In conclusion, the implementation of this user authentication system is a strategic investment that will enhance the security and usability of the web application, leading to increased user satisfaction and potentially higher user retention rates.
Route: /login/test-token
Functions: No functions

Route: /password-recovery/{email}
Functions:
- `get_user_by_email`
- `generate_password_reset_token`
- `generate_reset_password_email`
- `send_email`

Route: /reset-password/
Functions:
- `verify_password_reset_token`
- `get_user_by_email`
- `get_password_hash`

Case:
Business Case:

The business case revolves around the development and implementation of a secure user management system for a web-based application. The system is designed to handle user authentication, password management, and user retrieval functionalities.

1. User Authentication: The system uses JWT tokens for user authentication. The function `verify_password_reset_token` is used to decode the provided token using the `jwt.decode()` method. If the decoding is successful, it extracts the value of the 'sub' (subject) field from the decoded object and returns it as a string. This function is part of the password reset functionality, where a token is generated and sent to the user via email or SMS, which can then be used to reset their password.

2. Password Management: The system securely stores user passwords by hashing them before storing in a database or file system. The function `get_password_hash` takes a password as an input parameter and returns its hash using the `pwd_context` module. By returning the hash instead of the

plaintext password, this function helps in mitigating potential security risks associated with storing user passwords in plain text format.

3. User Retrieval: The system retrieves user information based on their email address. The function `get_user_by_email` takes a SQLAlchemy session and an email as arguments. It returns either a `User` object or `None` after searching for the user with the given email using a SQLAlchemy query with the `select()` function.

The implementation of this system will enhance the security of user data, streamline user management processes, and improve the overall user experience. It will also help the organization comply with data protection regulations and standards.
Route: /password-recovery-html-content/{email}
Functions:

- `get_user_by_email`
- `generate_password_reset_token`
- `generate_reset_password_email`

Case:
Business Case:

The business case revolves around the development and implementation of a secure and efficient password reset system for a web-based application. The system is designed to enhance user experience and security by providing a streamlined process for users to reset their passwords.

1. **Problem Statement**: Users of the web-based application need a secure and efficient way to reset their passwords when they forget them or want to change them for security reasons. The current process is manual and time-consuming, leading to a poor user experience and potential security risks.

2. **Proposed Solution**: Implement a password reset system that allows users to reset their passwords through their registered email addresses. The system will include functions to retrieve user information based on email, generate a secure password reset token, and send a password reset email to the user.

3. **Benefits**:
- **Improved User Experience**: The automated password reset process will be faster and more convenient for users, improving their overall experience with the application.
- **Enhanced Security**: The use of secure tokens for password reset will reduce the risk of unauthorized access to user accounts.
- **Reduced Operational Costs**: Automating the password reset process will reduce the need for manual intervention, saving time and resources.

4. **Implementation Plan**: The system will be implemented using Python and SQLAlchemy for database operations, JWT for token generation, and Django for email template rendering. The implementation will involve the following functions:
- `get_user_by_email`: Retrieves a user from the database using their email address.
- `generate_password_reset_token`: Generates a secure token for password reset.
- `generate_reset_password_email`: Creates a password reset email with a link containing the token.

5. **Cost and Time Estimates**: The development and implementation of the system are expected to take around 2-3 weeks, considering testing and deployment. The cost will depend on the development resources required and the complexity of integrating the system into the existing application.

6. **Risk Assessment**: Potential risks include users not receiving the password reset email due to spam filters or other email issues, and users' email accounts being compromised, leading to unauthorized password resets. These risks will be mitigated by implementing robust email delivery systems and educating users about account security.

7. **Success Metrics**: The success of the system will be measured by the reduction in manual password reset requests, the decrease in time taken for users to reset their passwords, and user feedback on the new process.

items.py

Route: /
Functions: No functions

Route: /{id}
Functions: No functions

Dataclasses overview

Based on your provided code snippet, it seems like you've defined a number of classes to represent various entities and operations within your FastAPI application. Let me provide some insight into each class:

1. `UserBase` - This is the base class for all user-related models. It includes properties such as email, is_active, is_superuser, and full_name (optional).
2. `UserCreate` - This class extends `UserBase` and adds a property for password. This class will be used to create new users.
3. `UserRegister` - This class extends `SQLModel` and includes properties for email, password, and full_name (optional). It will be used when registering new users.
4. `Token` - This is not shown in your provided code snippet, but based on the naming convention, it's possible that this class represents a JWT token generated by FastAPI's OAuth2 or Token authentication schemes.
5. `EmailSettings` - This class holds various email-related settings such as SMTP server host, port, and encryption method (TLS/SSL), SMTP user, password, test user, and emails enabled flag.
6. `Settings` - This is the main settings class for your entire FastAPI application. It includes a number of properties such as the project name, first superuser username and password, secret keys, Sentry DSN (optional), email reset token expiry period, and users open registration flag.

These classes can be used to interact with your database, generate JWT tokens, and send emails using FastAPI's built-in email functionality or an external SMTP server. By defining these models in this way, you can create a clear separation between your application's core business logic and its data access and communication layers. This approach also facilitates better testability and code maintainability since changes to the underlying database schema or communication protocols won't affect your application's higher-level business logic.

Dataclasses

UserBase

```
from pydantic import BaseModel, Field, SQLModel

class UserBase(SQLModel):
""" Unique email for the user"""
email: str = Field(unique=True, index=True)
""" By default, the user is active and not a superuser"""
is_active: bool = True
is_superuser: bool = False
""" Optional full name of the user"""
full_name: str | None = None
```

UserCreate

Here's the updated code with comments explaining each field:

```
from pydantic import BaseModel, Field

""" This class is used to create a new user account"""
class UserCreate(BaseModel):
""" This field is required and contains the password for the new user"""
password: str = Field(min_length=8)
```

I've added comments explaining each field. Let me know if you need any further assistance!

UserRegister

Adding comments to explain the purpose of each field in the `UserRegister` dataclass using Pydantic's SQLModel:

```
from pydantic import BaseModel, Field, EmailStr, constr
from typing import Optional
from sqlalchemy.orm import SQLModel

class UserRegister(SQLModel):
email: str = Field( ... ) # Required and must be a valid email address
password: str = Field( ... ) # Required password
full_name: Optional[str] = None # Optional full name for user, default is None
```

```

Explanation of the comments added:
- `BaseModel` is a base class for Pydantic models, which provides various features such as schema validation, input/output transformations, and more. `SQLModel` is a SQLAlchemy subclass of `BaseModel`, which adds database persistence support.
- `Field()` is a decorator used to add fields to a dataclass, and can be used to specify the field's name, type, and validation rules. The `...` argument represents that this field accepts any type as input.
- `EmailStr` is a custom validator provided by Pydantic which validates email addresses based on RFC 5322 specifications. This ensures that email addresses in the `UserRegister` model are properly formatted.
- `constr()` is a Pydantic decorator used to define custom constraints for fields, such as minimum/maximum length or regular expressions. It's not being used here, but it could be added if required.
- `Optional[str]` is a type annotation for the `full_name` field, which specifies that this field is optional and can accept a string value. If we wanted to make it required, we would remove the `None` from the annotation.

## UserUpdate

To add comments to this Pydantic model, you can wrap each field with a comment explaining its purpose:

```
from pydantic import BaseModel
from typing import Optional

class UserUpdate(BaseModel):
"""
Update user information.
"""

email: Optional[str] = None
"""
New email address for the user (none to keep old).
"""

password: Optional[str] = None
"""
New password for the user (none to keep old).
"""
```

Alternatively, you can provide a docstring at the class level:

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

```
class UserUpdate(BaseModel):
"""
Update user information.
"""

email: Optional[str] = None
password: Optional[str] = None
```

In this case, you would provide a docstring at the class level that describes what the `UserUpdate` model is used for. This can be helpful when working in a team or with collaborators who may not be familiar with the specifics of the model.

UserUpdateMe

To add comments to the Pydantic model definition, you can wrap each field and its type with a comment explaining its purpose:

```
from pydantic import BaseModel, SQLModel
from typing import Optional

class UserUpdateMe(SQLModel):
"""
Model for updating user information.
"""

full_name: Optional[str] = None # Optional field to update the user's full name.
email: Optional[str] = None # Optional field to update the user's email address.
```

Here, each field and its type are followed by a comment explaining its purpose. These comments will help other developers understand the intended usage of the model and the expected input values for each field.

UpdatePassword

Here's the updated code with comments added:

```
from pydantic import BaseModel, Field, SQLModel
from typing import Annotated

class UpdatePassword(SQLModel):
"""
Model to update user's password.
```

```
Attributes:
current_password (str): Current password of the user. Required.
new_password (str): New password to be set for the user. Required.
"""
current_password: str = Field(..., min_length=8, max_length=64)
new_password: str = Field(..., min_length=8, max_length=64)

""" Using SQLModel and Annotated decorator for database usage."""
class User(SQLModel):
id: int = Field(default=None, primary_key=True)
username: str = Field(unique=True)
email: str = Field(unique=True)
password_hash: str = Field()

""" Using Pydantic model for request/response."""
class LoginForm(BaseModel):
"""
Model to authenticate user's credentials.

Attributes:
username (str): Username of the user. Required.
password (str): Password provided by the user. Required.
"""
username: str = Field(..., min_length=3, max_length=64)
password: str = Field(..., min_length=8, max_length=64)
```

User

```
from pydantic import BaseModel, Field, HttpUrl
from sqlalchemy.orm import Mapped, mapped_column, relationship
from typing import List

class UserBase(BaseModel):
""" This is the base class for all user models, which includes common
fields that are required in all user data."""
email: str
password: str

""" This is the main User model that inherits from UserBase and adds a
primary key (id) and a foreign key relationship (items) to the Item
model."""
class User(UserBase, table=True):
id: int | None = Field(default=None, primary_key=True)
hashed_password: str

""" A one-to-many relationship with the Item model, which is represented
as a list of Items that belong to this User. The back_populates argument
specifies that the 'owner' attribute should be populated when querying
```

```
the Item model with this User as the filter."""
items: List['Item'] = Relationship(back_populates='owner')

""" A helper function for converting an HTTP URL to a SQLAlchemy
Mapped[HttpUrl] type, which can then be used in the dataclass
definition."""
@classmethod
def from_http_url(cls, http_url: HttpUrl) -> Mapped['User']:
""" This function is useful when converting HTTP URLs passed as
parameters to a database model that requires SQLAlchemy Mapped[HttpUrl]
types."""
return cls(**http_url.dict())
```

UserPublic

To add comments explaining the purpose and meaning of each attribute in the `UserPublic` class, you can modify the code as follows:

```
from pydantic import BaseModel

class UserBase(BaseModel):
""" This is the base model for all user-related data. It contains common
attributes such as email and password."""
email: str
password: str

class UserPublic(UserBase):
id: int
""" This class represents a public view of a user object. It includes the
unique ID (`id`) which can be used to retrieve other details about the
user using an API endpoint."""
```

In this modified code, comments are added to explain the purpose and meaning of each attribute in both `UserBase` and `UserPublic` classes. This helps other developers better understand the data structures being used in your FastAPI application.

UsersPublic

```
from pydantic import BaseModel, Field
from typing import List

class UserPublic(BaseModel):
""" This is a public representation of the User model."""
""" It should contain all fields that can be shared externally without
revealing any sensitive information."""
id: int = Field(..., description="Unique identifier for the user.")
```

```
email: str = Field(..., description="User's email address.",
min_length=5, max_length=254)
username: str = Field(..., description="User's preferred username.")
first_name: str = Field(..., description="User's first name.")
last_name: str = Field(..., description="User's last name.")

class UsersPublic(BaseModel):
""" This is a list of public representations of User models."""
data: List[UserPublic] = Field(...)
count: int = Field(..., description="Number of users in the list.")
```
In this example, we've added comments to each field and the `UsersPublic` class as a whole to explain their purpose and any relevant constraints. These comments can be used by clients or other developers to better understand how to use these classes.

ItemBase

To add comments to the Pydantic model, you can modify it as follows:

```
from pydantic import BaseModel, SQLModel
from typing import Optional

class ItemBase(SQLModel):
"""Represents a base model for an item."""

title: str

description: Optional[str] = None

class Config:
"""Configures the Pydantic settings"""
orm_mode = True
```

The comments explain what each part of the code does and why it's necessary. The `Config` attribute is also added, which configures some settings for the SQLModel. This can be useful for optimizing performance when working with a database.

Here's how you might use this model in your FastAPI app:

```
from fastapi import FastAPI
from models import ItemBase

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int, item: ItemBase = Depends()):
```

```
    """Returns a specific item by ID."""
    return item
```

In this example, the `ItemBase` model is used as a dependency for the `read_item` endpoint. This allows you to validate and sanitize the input data using Pydantic's built-in features.

ItemCreate

```
from pydantic import BaseModel

class ItemCreate(BaseModel):
"""
Model for creating new items.
"""
title: str
```

Explanation:
- We are inheriting `ItemCreate` class from the base model provided by Pydantic `BaseModel`.
- Added docstring to explain what this model is used for. This will be useful while working with other developers and also for future reference.

ItemUpdate

Here's the same Pydantic dataclass with comments added for clarity:

```
from pydantic import BaseModel

class ItemBase(BaseModel):
""" This is a base class for all items, containing common fields."""
name: str
price: float
description: str | None = None

""" Overriding the __init__ method to set default values for some fields."""
def __init__(self, *args, **kwargs):
super().__init__(*args, **kwargs)
self.title = None

class ItemUpdate(ItemBase):
""" This is a subclass of ItemBase that can be used to update existing items."""
title: str | None = None
```

```
""" This field is optional and can hold None or a string value."""
```

In this example, I've added comments explaining the purpose of each class and method, as well as comments for any unusual behavior. Feel free to modify it based on your specific use case!

Item

```
from pydantic import BaseModel, Field, Sqlite3Dialect
from typing import Optional

class ItemBase(BaseModel):
title: str

class Item(ItemBase, table=True):
""" Set primary key to 'id' and make it optional by default"""
id: int | None = Field(default=None, primary_key=True)
""" Set foreign key relationship with 'User' model for the 'owner_id' field"""
owner_id: int | None = Field(default=None, foreign_key='user.id', nullable=False)
""" Define a relationship between this model and another model called 'User'"""
""" This will be used to automatically populate the 'owner' field of this model with data from the related 'User' model"""
owner: "User" | None = Relationship(back_populates='items')

""" Set the SQLite3Dialect for this datamodel"""
Item.metadata.sqlite_dialect = Sqlite3Dialect()
```

ItemPublic

```
from pydantic import BaseModel

class ItemPublic(BaseModel):
""" `id` is the unique identifier for this item"""
id: int

""" `owner_id` identifies the user who owns this item"""
owner_id: int
```

ItemsPublic

```
from pydantic import BaseModel, Field
from typing import List

class ItemPublic(BaseModel):
"""
This is the public representation of an item object.
"""
name: str = Field(...)
description: str | None = None
price: float = Field(ge=0.0)

class ItemsPublic(BaseModel):
"""
A list of public representations of item objects, along with a count.
"""
data: List[ItemPublic] = Field(...)
count: int = Field(ge=0, le=1000)
```

The comments added here provide more context and clarity to the code for other developers who may be working on this project. They explain what each model represents and what type of data it contains. This makes the code more self-documenting and easier to understand at a glance. It also helps to enforce consistency in naming conventions and data types across models.

Message

To add comments to the above Pydantic model, you can modify it as follows:

```
from pydantic import BaseModel, SQLModel
from sqlalchemy import Integer, String
from typing import Optional

class Message(SQLModel):
"""
This is a message class used for messaging in the application.
Attributes:
- message (str): The actual message content.
"""
message: str = None

class Config:
orm_mode = True
```

Explanation:
- `from pydantic import BaseModel, SQLModel` imports the necessary classes from Pydantic.

- `from sqlalchemy import Integer, String` imports the data types used in the database for integer and string fields respectively.
- `from typing import Optional` imports the `Optional` type hint to specify that a field can be nullable.
- The class is defined as a subclass of `SQLModel`, which provides database connectivity.
- A docstring is added above the class definition to provide a description and details about the class.
- Inside the class, the `message` attribute is defined with an initial value of `None`.
- The `orm_mode = True` property of the `Config` class is set to enable SQLAlchemy compatibility.

Note: In this example, `Integer` and `String` types are not explicitly used for their corresponding attributes as we're inheriting from `SQLModel`, which provides these fields automatically based on the model's attributes. However, if you want more fine-grained control over these field types, you can replace these with their respective SQLAlchemy data types in the inheritance hierarchy of `BaseModel`.

Token

Here's the updated version of the `Token` dataclass with comments explaining each field and its purpose:

```
from pydantic import BaseModel, Field
from sqlalchemy.orm import SQLModel

class Token(SQLModel):
""" The access token used to authenticate the user"""
access_token: str
""" This is set to 'bearer' by default because it's the most commonly used token type for authorization headers."""
token_type: str = Field(default='bearer')
```

Alternatively, you can also add comments in a docstring at the class level, like this:

```
from pydantic import BaseModel, Field
from sqlalchemy.orm import SQLModel

class Token(SQLModel):
"""A token object used for authentication."""

access_token: str
# This is set to 'bearer' by default because it's the most commonly used token type for authorization headers.
token_type: str = Field(default='bearer')
```

```

Either way, the comments should help make the intent and purpose of each field clearer.

## TokenPayload

To add comments to the `TokenPayload` class, you can modify it as follows:

```
from pydantic import BaseModel, SQLModel
from typing import Optional

class TokenPayload(SQLModel):
"""
Represents a JWT payload.

Attributes:
sub (int | None): The subject ID. Can be `None`.
"""
sub: Optional[int] = None
```

In this updated version, I added a docstring to the class that explains what it represents and lists its attributes with descriptions.

## NewPassword

```
from pydantic import BaseModel, Field
from typing import Optional

class NewPassword(BaseModel):
"""
Model for resetting a password using a token.
"""

token: str = Field(..., description="Reset token issued by the server.")
new_password: str = Field(..., min_length=8, description="The new password to set.")

class Config:
orm_mode = True
schema_extra = {
"example": {
"token": "1234567-89ab",
"new_password": "super_secure_password"
}
}
```
```

I added comments to explain the purpose and usage of each field. The `schema_extra` attribute is used to provide a JSON schema for this model, including an example value. This is useful for generating client-side code or documentation.

To convert this Pydantic dataclass into a SQLAlchemy ORM model (SQLModel), you can use the `orm_mode = True` configuration option in the `Config` class.

EmailData

Adding comments to explain each field in the EmailData dataclass defined using Pydantic:

```
@dataclass
class EmailData:
"""
This is a Pydantic dataclass representing data required for sending an email.
"""

html_content: str = None
"""
The content of the email in HTML format. Required.
"""

subject: str = None
"""
The subject line of the email. Required.
"""
```

Settings

Here's the commented version of your Pydantic dataclass:

```
from pydantic import BaseSettings, HttpUrl, PostgresDsn, SecretStr, settings
from pydantic.env_settings import SettingsSourceCallable
from typing import Annotated, Literal

class Settings(BaseSettings):
""" Load environment variables from .env and ignore empty ones"""
env_file = '.env'
env_source = SettingsSourceCallable(settings.getenv)
env_ignore_empty = True
```

```
""" Define settings with default values"""
model_config: SettingsConfigDict = ...
API_V1_STR: str = '/api/v1'
SECRET_KEY: SecretStr = SecretStr(key_derivation='hmac-sha256')
ACCESS_TOKEN_EXPIRE_MINUTES: int = 60 * 24 * 8
DOMAIN: str = 'localhost'
ENVIRONMENT: Literal['local', 'staging', 'production'] = 'local'

""" Define a computed field that combines the domain and environment"""
@computed_field
@property
def server_host(self) -> str:
if self.ENVIRONMENT == 'local':
return f'http://{self.DOMAIN}'
return f'https://{self.DOMAIN}'

""" Define a list of allowed origins for CORS"""
BACKEND_CORS_ORIGINS: Annotated[list[AnyUrl] | str,
BeforeValidator(parse_cors)] = []
PROJECT_NAME: str
SENTRY_DSN: HttpUrl | None = None
POSTGRES_SERVER: str
POSTGRES_PORT: int = 5432
POSTGRES_USER: str
POSTGRES_PASSWORD: str
POSTGRES_DB: str = ''

""" Define a computed field that generates the SQLAlchemy database URI
based on the settings"""
@computed_field
@property
def SQLALCHEMY_DATABASE_URI(self) -> PostgresDsn:
return MultiHostUrl.build(
scheme='postgresql+psycopg',
username=self.POSTGRES_USER,
password=self.POSTGRES_PASSWORD,
host=self.POSTGRES_SERVER,
port=self.POSTGRES_PORT,
path=self.POSTGRES_DB,
)

""" Define a BeforeValidator function to validate the SMTP settings"""
SMTP_TLS: bool = True
SMTP_SSL: bool = False
SMTP_PORT: int = 587
SMTP_HOST: str | None = None
SMTP_USER: str | None = None
SMTP_PASSWORD: str | None = None
EMAILS_FROM_EMAIL: str | None = None
EMAILS_FROM_NAME: str | None = None

""" Define a computed field that enables/disables email functionality
```

```
based on the SMTP settings"""
@computed_field
@property
def emails_enabled(self) -> bool:
return bool(self.SMTP_HOST and self.EMAILS_FROM_EMAIL)

""" Define an email reset token expiration time in hours"""
EMAIL_RESET_TOKEN_EXPIRE_HOURS: int = 48

""" Define the test user for email functionality"""
EMAIL_TEST_USER: str = 'test@example.com'

""" Define the superuser credentials and enable/disable open registration
based on settings"""
FIRST_SUPERUSER: str
FIRST_SUPERUSER_PASSWORD: str
USERS_OPEN_REGISTRATION: bool = False

""" Validate non-default secrets with a BeforeValidator function"""
@model_validator(mode='after')
def _enforce_non_default_secrets(self) -> Self:
self._check_default_secret('SECRET_KEY', self.SECRET_KEY)
self._check_default_secret('POSTGRES_PASSWORD', self.POSTGRES_PASSWORD)
self._check_default_secret('FIRST_SUPERUSER_PASSWORD',
self.FIRST_SUPERUSER_PASSWORD)

""" Define a helper function to check for default secrets and raise an
exception if found"""
def _check_default_secret(self, name: str, secret: SecretStr):
if secret.get_secret_value() == SecretStr().get_secret_value():
raise ValueError(f'{name} should not be set to the default value')
```

get_user_by_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/crud.py

This Python function, named `get_user_by_email`, takes in a SQLAlchemy session and an email as arguments, marked with the `*` decorator to indicate keyword-only parameters. It returns either a `User` object or `None` after searching for the user with the given email using a SQLAlchemy query with the `select()` function. The user is retrieved from the session using `exec()`, and its value is assigned to the variable `session_user`. Finally, the `return` statement sends back either the found `User` or `None`. In summary, this function allows finding a specific user based on their email address in SQLAlchemy.

````
The system will generate the following commented code based on the
analysis of the function:
```
def get_user_by_email(*, session: Session, email: str) -> User | None:
"""
Returns the user with the given email address, or None if no such user
exists.

Parameters:
-----------
* `session`: The SQLAlchemy session object to use for querying the
database.
- `email`: The email address of the user to retrieve.

Returns:
--------
A User object if a user with the given email exists, or None otherwise.
"""
statement = select(User).where(User.email == email)
session_user = session.exec(statement).first()
return session_user
```
````

create_user

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/api/routes/users.py

The Python function `create_user` creates a new user when called with arguments `session` and `user_in`. If a user with the same email address already exists in the system, an HTTPException is raised. Otherwise, the `crud` module's `create_user` function is invoked to add the new user to the database. If email notifications are enabled and a valid email address was provided for the new user, a new account confirmation email is generated using the `generate_new_account_email` function and sent to the email address provided by the user. The completed user object is then returned.

```
The function `create_user` takes in two parameters: `session` and
`user_in`. The `session` parameter is of type `SessionDep`, while the
```

`user_in` parameter is of type `UserCreate`. The function returns an `Any` value.

Here's a commented version of the function based on the analysis:
```
def create_user(
""" Session object used for CRUD operations"""
session: SessionDep,

""" User creation input data"""
user_in: UserCreate,
):
"""
Create new user.
"""
""" Check if a user with the same email already exists in the system"""
user = crud.get_user_by_email(session=session, email=user_in.email)
if user:
raise HTTPException(
status_code=400,
detail="The user with this email already exists in the system.",
)

""" Create new user and return it"""
user = crud.create_user(session=session, user_create=user_in)
if settings.emails_enabled and user_in.email:
""" Generate email data for the new account"""
email_data = generate_new_account_email(
email_to=user_in.email, username=user_in.username,
password=user_in.password
)

""" Send email to new user with account details"""
send_email(
email_to=user_in.email,
subject=email_data.subject,
html_content=email_data.html_content,
)

return user
```

generate_new_account_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/utils.py

The Python function `generate_new_account_email` takes three arguments `email_to`, `username`, and `password`. It generates a new account email with the given information and returns an object called `EmailData` containing the HTML content and subject of the email. The email's subject is formatted using string formatting, where `project_name` comes from the `settings` module, and the username is passed as an argument. A template named `new_account.html` is rendered with the

given context, which includes variables like `project_name`, `username`, `password`, `email`, and `server_host`. This function can be used to automatically send a new account confirmation email to the user's registered email address.

````
The system will parse the function and generate a commented version of it
based on its analysis. The generated code will be similar to the
following:
```
"""" Function that generates new account email data"""
def generate_new_account_email(
""" Email address to send the email to"""
email_to: str,
""" Username for the new account"""
username: str,
""" Password for the new account"""
password: str
) -> EmailData:
"""
This function generates a new account email data object.

Parameters:
email_to (str): Email address to send the email to.
username (str): Username for the new account.
password (str): Password for the new account.

Returns:
EmailData: An email data object with the generated email content and
subject.
"""
""" Load project settings"""
settings = load_settings()

""" Set the project name"""
project_name = settings.PROJECT_NAME

""" Create the subject of the email"""
subject = f"{project_name} - New account for user {username}"

""" Render the email template with the given context"""
html_content = render_email_template(
template_name="new_account.html",
context={
"project_name": settings.PROJECT_NAME,
"username": username,
"password": password,
"email": email_to,
"link": settings.server_host,
},
)

""" Return the email data object with the generated content and
subject"""
````
````

```
return EmailData(html_content=html_content, subject=subject)
```

The system will also provide a detailed explanation of each parameter, including its purpose, type, and default value.

send_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/utils.py

This Python function, named `send_email`, allows sending an email with a customizable subject and body. The function takes in three arguments: the required recipient email address `email_to`, and optional arguments for the email subject (default is empty) and HTML content for the body.

The function starts by checking if there are provided configurations for email variables using the `settings` module. If not, it raises an assertion error. It then creates a new message object with the specified subject and HTML content. The sender name and email address are set from the configurations in the `settings` module.

The SMTP options are configured based on settings for TLS, SSL, host, and port, before the message is sent to the recipient using the `Message.send()` method with the specified smtp options. The function then logs the result of sending the email. Overall, this function provides a convenient way to send customizable emails from a Python application while still allowing configuration flexibility.

```
The system will generate a commented version of the function based on its analysis of the input code. Here is an example of what the commented version of the `send_email` function might look like:
```
```
def send_email(
""" Send email with given parameters"""
*,
email_to: str, # Email address to send email to
subject: str = "", # Subject line of the email (optional)
html_content: str = "", # HTML content of the email (optional)
) -> None:
assert settings.emails_enabled, "no provided configuration for email variables" # Check if emails are enabled in the current environment
message = emails.Message(
subject=subject,
html=html_content,
mail_from=(settings.EMAILS_FROM_NAME, settings.EMAILS_FROM_EMAIL),
) # Create a new email message with the given parameters
smtp_options = {"host": settings.SMTP_HOST, "port": settings.SMTP_PORT}
# Set up SMTP options for sending the email
if settings.SMTP_TLS: # Check if TLS is enabled for SMTP
smtp_options["tls"] = True
elif settings.SMTP_SSL: # Check if SSL is enabled for SMTP
smtp_options["ssl"] = True
if settings.SMTP_USER: # Check if a username is provided for SMTP authentication
smtp_options["user"] = settings.SMTP_USER
```

```
if settings.SMTP_PASSWORD: # Check if a password is provided for SMTP
authentication
smtp_options["password"] = settings.SMTP_PASSWORD
response = message.send(to=email_to, smtp=smtp_options) # Send the email
using the given SMTP options
logging.info(f"send email result: {response}") # Log the result of
sending the email
```
This commented version includes explanatory comments for each line of code and helps users understand what the function does and how it works.

get_password_hash

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/core/security.py

This Python function, named `get_password_hash`, takes a password as an input parameter of type string and returns its hash using the `pwd_context` module. The returned value is also of type string. This function's purpose is to securely store user passwords by hashing them before storing in a database or file system.

The `pwd_context` module is a part of the Unix cryptography libraries and is available as a Python library named 'crypt'. It provides methods for password storage using industry-standard algorithms like SHA-256, BCRYPT, and PBKDF2HMACSHA1.

By returning the hash instead of the plaintext password, this function helps in mitigating potential security risks associated with storing user passwords in plain text format.

```
The system will parse the function and identify its purpose, parameters,
and return value. Based on this analysis, it will generate a commented
version of the function like this:
```
```
def get_password_hash(password: str) -> str:
"""
Generates a hashed password using the pwd_context object.

:param password: The plaintext password to be hashed.
:return: The hashed password as a string.
"""
return pwd_context.hash(password)
```
```
The commented version includes a summary of the function's purpose, a
description of each parameter and its type, and a list of any exceptions
that may be raised by the function. This information can be used to help
other developers understand the function's behavior and usage, and to
ensure that the function is used correctly and safely within the
codebase.
```

get_user_by_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/crud.py

The function `get_user_by_email` in Python, given an email and a database session, retrieves the user with the matching email from the database using SQLAlchemy's select statement. The returned value can be either the found user object or None if no user is found. The `*` argument syntax is used to indicate that the function can take keyword arguments, but in this specific case, it's not required as all parameters are present. This function facilitates quick and efficient retrieval of users based on their email addresses without needing to write complex SQL queries.

```
Here's the commented version of the function based on the analysis of the
user input:
```
```
""" Retrieves a user by their email address"""
def get_user_by_email(*, session: Session, email: str) -> User | None:
"""
Returns the user with the specified email address if found, otherwise
returns None.

Parameters:
- session (Session): The SQLAlchemy session to use for querying the
database.
- email (str): The email address of the user to retrieve.

Return value:
- User | None: The retrieved user object, or None if no user with the
specified email address was found.
"""
statement = select(User).where(User.email == email)
session_user = session.exec(statement).first()
return session_user
```

create_user

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/api/routes/users.py

The Python function `create_user` creates a new user using the `crud` module's `create_user` function, provided the email address of the new user is not already in use. If emails are enabled and an email address has been provided for the new user, a new account email will be generated and sent via the `send_email` function. The function returns the newly created user object. In summary, this function allows for the creation of a new user with optional email verification.

```
The system will analyze the function and generate a commented version of
it based on its purpose, parameters, and return value. Here's an example
of what the commented code might look like:
```
```
```

```
""" create_user(session: SessionDep, user_in: UserCreate) -> Any"""
def create_user(*, session: SessionDep, user_in: UserCreate) -> Any:
"""
Create new user.

Parameters:
-----------
session: The database session object used for interacting with the
database.
user_in: A dictionary containing information about the user to be
created.

Returns:
--------
Any: The created user object, or None if an error occurred.

Raises:
-------
HTTPException(400): If a user with the same email already exists in the
system.
"""
""" Check if a user with the given email already exists"""
user = crud.get_user_by_email(session=session, email=user_in.email)
if user:
raise HTTPException(
status_code=400,
detail="The user with this email already exists in the system.",
)

""" Create the new user"""
user = crud.create_user(session=session, user_create=user_in)

""" Send a welcome email if emails are enabled and the user's email is
set"""
if settings.emails_enabled and user_in.email:
email_data = generate_new_account_email(
email_to=user_in.email, username=user_in.email,
password=user_in.password
)
send_email(
email_to=user_in.email,
subject=email_data.subject,
html_content=email_data.html_content,
)

return user
```
Note that the commented code includes information about the function's parameters and return value, as well as any exceptions that it may raise. It also provides a brief description of the function's purpose and its expected inputs and outputs.

get_password_hash

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/core/security.py

The given Python function is named `get_password_hash` and takes a string argument `password`. It then uses the `pwd_context` to hash the password using the `hash()` method and returns the hashed result as a string. This function seems to be used for securely storing passwords by hashing them instead of storing plain text passwords.

````
The system parses the function to identify its purpose, parameters, and
return value. Based on this analysis, it generates a commented version of
the function as follows:
```
def get_password_hash(password: str) -> str: # Hashes the input password
using the pwd_context object
"""
Purpose: Returns a hashed version of the input password.

Parameters:
* password (str): The input password to be hashed.

Return value:
A str representing the hashed password.
"""
return pwd_context.hash(password)
```
The system comments on the purpose of the function, the parameters it
takes in, and the return value it produces. It also provides a detailed
explanation of what the function does and how it works.
````

generate_test_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/utils.py

This Python function named `generate_test_email` takes an email address (stored in the variable `email_to`) as input and returns an object of type `EmailData`. The function defines two variables `project_name` and `subject` which are initialized using the `settings.PROJECT_NAME` variable. The `html_content` is generated by rendering a template called 'test_email.html' with context values including 'project_name' and 'email'. The `EmailData` object is then returned, containing the generated HTML content and subject for the email. In summary, this function generates an email with a test message and sends it to a specified email address.

```
[PYTHON]
def generate_test_email(email_to: str) -> EmailData:
"""
Generate a test email for the given recipient.

Parameters:
email_to (str): The recipient's email address.
```

```
Returns:
EmailData: An object containing the email data.
"""
project_name = settings.PROJECT_NAME
subject = f"{project_name} - Test email"
html_content = render_email_template(
template_name="test_email.html",
context={"project_name": settings.PROJECT_NAME, "email": email_to},
)
return EmailData(html_content=html_content, subject=subject)
[/PYTHON]

The system will generate the commented version of the function as
follows:
```
```
def generate_test_email(email_to: str) -> EmailData:
"""
Generate a test email for the given recipient.

Parameters:
email_to (str): The recipient's email address.

Returns:
EmailData: An object containing the email data.
"""
```
The system will also provide the user with the commented code, which they can use to understand the purpose and parameters of the function, as well as its return value.

send_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/utils.py

The function `send_email` in Python is used to send an email with a customizable subject and optional HTML content to the specified email address. The configuration for email variables must be provided, and the function raises an assertion error if no configuration is available. The email is sent using the settings for SMTP server, port, TLS/SSL encryption, username, and password. After sending the email, a log message containing the result of the email send operation is generated. The default value for subject is an empty string ("").

```
The system will generate a commented version of the function based on its
analysis. Here's an example of what the commented code might look like:
```
```
def send_email(
""" *,"""
""" email_to: str,"""
""" subject: str = "","""
""" html_content: str = "","""
```

```
) -> None:
"""Sends an email with the specified parameters.

Args:
email_to (str): The recipient's email address.
subject (str, optional): The subject line of the email. Defaults to "".
html_content (str, optional): The HTML content of the email. Defaults to
"".

Returns:
None: No return value.

Raises:
AssertionError: If the configuration for email variables is not provided.
"""
""" Check if email variables are configured"""
assert settings.emails_enabled, "no provided configuration for email
variables"

""" Create an email message"""
message = emails.Message(
subject=subject,
html=html_content,
mail_from=(settings.EMAILS_FROM_NAME, settings.EMAILS_FROM_EMAIL),
)

""" Set the SMTP options"""
smtp_options = {"host": settings.SMTP_HOST, "port": settings.SMTP_PORT}
if settings.SMTP_TLS:
smtp_options["tls"] = True
elif settings.SMTP_SSL:
smtp_options["ssl"] = True
if settings.SMTP_USER:
smtp_options["user"] = settings.SMTP_USER
if settings.SMTP_PASSWORD:
smtp_options["password"] = settings.SMTP_PASSWORD

""" Send the email using the SMTP options"""
response = message.send(to=email_to, smtp=smtp_options)

""" Log the result of sending the email"""
logging.info(f"send email result: {response}")
```

authenticate

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/crud.py

The `authenticate` function in Python takes two arguments, `session` and `email`, as well as optional keyword arguments `password` and returns a `User` object or `None`. The function retrieves the user with

the given email from the database using `get_user_by_email()` and checks if their password is correct using `verify_password()`. If either check fails, the function returns `None`, indicating an unsuccessful login attempt. Otherwise, it returns the `User` object. This function seems to be responsible for authenticating a user based on their email and password.

```
```
def authenticate(*, session: Session, email: str, password: str) -> User | None:
"""Authenticates a user using their email and password.

Args:
session (Session): The current session.
email (str): The user's email address.
password (str): The user's password.

Returns:
User | None: The authenticated user, or None if authentication failed.
"""
db_user = get_user_by_email(session=session, email=email)
if not db_user:
return None
if not verify_password(password, db_user.hashed_password):
return None
return db_user
```
```

create_access_token

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/core/security.py

The Python function `create_access_token` takes in a subject and an expiration time duration as arguments and generates a JSON Web Token (JWT) using the PyJWT library. The JWT is encoded with the secret key provided by the `settings` module, and includes the subject and expiration time in its payload. The function returns the generated JWT as a string.

```
Here is the commented version of the function:
```
def create_access_token(subject: str | Any, expires_delta: timedelta) -> str:
"""
This function creates an access token for a given subject and expiry time.

Args:
subject (str | Any): The subject of the access token. Can be any type that is compatible with the `str` type.
expires_delta (timedelta): The duration after which the access token should expire.

Returns:
str: The encoded JWT token for the given subject and expiry time.
```
```

```
"""
""" Generate a UTC timestamp for the expiration of the access token"""
expire = datetime.utcnow() + expires_delta

""" Create a dictionary to encode with the JWT library"""
to_encode = {"exp": expire, "sub": str(subject)}

""" Encode the dictionary with the JWT library and return the encoded
token"""
encoded_jwt = jwt.encode(to_encode, settings.SECRET_KEY,
algorithm=ALGORITHM)
return encoded_jwt
```

get_user_by_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/crud.py

The Python function `get_user_by_email` takes a `session` object and an email address as arguments. It uses the SQLAlchemy `select` function to retrieve the user with the given email from the database, stored in the `User` class. The returned value is either the retrieved `User` object or `None` if no user was found. This function can be used to quickly and efficiently search for a specific user based on their email address without the need to write a more complex SQL query.

```
```
""" Function to retrieve a user by their email address"""
def get_user_by_email(*, session: Session, email: str) -> User | None:
""" This function takes in a SQLAlchemy session and an email address as
input"""
""" It returns the user object if found, or None otherwise"""

""" Statement to select the user with the given email address from the
database"""
statement = select(User).where(User.email == email)

""" Execute the statement on the SQLAlchemy session to retrieve the user
object"""
session_user = session.exec(statement).first()

return session_user
```
```

verify_password

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/core/security.py

The `verify_password` function in Python takes two arguments, `plain_password` (a plaintext password) and `hashed_password` (a hash value of a previously stored password), both of which are

passed as strings. This function then uses an instance of the `pwd_context` class to verify whether the `plain_password` matches the `hashed_password`. If the passwords match, the function returns `True`, indicating that the user has entered the correct password; otherwise, it returns `False`. The `pwd_context` class is a part of the standard library's `cryptography.fernet` module and provides methods for working with symmetric-key cryptography.

```
```
""" Verify password function"""
"""""""
""" Purpose: This function verifies if a plain password matches the
stored hashed password"""
"""""""
""" Parameters:"""
""" * plain_password (str): The plain password that needs to be
verified"""
""" * hashed_password (str): The stored hashed password that needs to be
matched with the plain password"""
"""""""
""" Return value:"""
""" * bool: Whether the plain password matches the stored hashed password
or not"""
def verify_password(plain_password: str, hashed_password: str) -> bool:
return pwd_context.verify(plain_password, hashed_password)
```
```

get_user_by_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/crud.py

The function `get_user_by_email` in Python is a method that retrieves a user with the specified email address from a database using the SQLAlchemy library's Session object. The function takes two arguments, `session` (a connection to the database) and `email` (the email address of the user to search for), both marked as keyword arguments (*).

The function starts by defining a query statement using the `select()` method from SQLAlchemy's Query object. This statement selects all columns from the `User` table where the `email` column matches the provided value. The statement is then executed using the `exec()` method, and the first result is returned as the `session_user` variable.

The function returns either the retrieved user (if found) or `None` if no user with the specified email address exists in the database.

```
Here is the commented version of the `get_user_by_email` function based
on the analysis:
```
```
def get_user_by_email(*, session: Session, email: str) -> User | None:
"""
Retrieves a user by their email address.

Parameters:
```

```
    session (Session): The database session to use for the query.
    email (str): The email address of the user to retrieve.

    Returns:
    User | None: The retrieved user, or None if no user with the given email
    address was found.
    """
    statement = select(User).where(User.email == email)
    session_user = session.exec(statement).first()
    return session_user
```

generate_password_reset_token

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/utils.py

This Python function generates a password reset token for a provided email address. It uses the timedelta function to set an expiration time of a specified number of hours, adds it to the current UTC time, and then calculates the timestamp of the new expiry date. The JWT (JSON Web Token) library is used to encode the email address, the expiry time, and a unique ID into a string token using the function's secret key and HS256 encryption algorithm. The generated token is returned. In short, this function creates a secure password reset token that can be sent to the user's email for them to use in resetting their password.

```
[PYTHON]
def generate_password_reset_token(email: str) -> str: # Function to
generate a password reset token for a given email address
delta = timedelta(hours=settings.EMAIL_RESET_TOKEN_EXPIRE_HOURS) #
Calculate the expiration time based on the number of hours specified in
settings
now = datetime.utcnow() # Get the current UTC time
expires = now + delta # Calculate the expiration date and time
exp = expires.timestamp() # Convert the expiration date and time to a
timestamp
encoded_jwt = jwt.encode( # Generate the JWT token
{"exp": exp, "nbf": now, "sub": email},
settings.SECRET_KEY,
algorithm="HS256",
)
return encoded_jwt # Return the encoded JWT token
[/PYTHON]
[TESTS]
""" Test case 1:"""
assert generate_password_reset_token("john.doe@example.com") ==
"encoded_jwt"
""" Test case 2:"""
assert generate_password_reset_token("jane.doe@example.com") ==
"encoded_jwt"
[/TESTS]
```

generate_reset_password_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/utils.py

The Python function `generate_reset_password_email` takes three arguments `email_to`, `email`, and `token`. It then uses the settings module to extract the project name and server host. The function constructs the email subject using these values, creates a link with the token and server host, and passes them along with some additional context to a template rendering function called `render_email_template`. Finally, it returns an `EmailData` object containing the generated HTML content and subject. Overall, this function generates a reset password email for the given user and email addresses, including a link with the token that expires after a certain number of hours specified in the settings.

```
The system will generate a commented version of the function based on its analysis. The commented code will include the following information:

1. Purpose: "Generates an email with instructions to reset a user's password."
2. Parameters:
* email_to (str): The recipient's email address.
* email (str): The user's email address.
* token (str): A token used for password recovery.
3. Return value:
* An object of type `EmailData`, containing the following attributes:
+ html_content (str): The HTML content of the email, including the reset password instructions.
+ subject (str): The subject line of the email, including the project name and user's email address.
```

send_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/utils.py

The Python function `send_email` sends an email using the `emails` library. It takes three arguments: `email_to` (required), `subject` (optional, default is empty string), and `html_content` (optional, default is empty string). The function first checks if there is a configuration provided for email variables using the `settings` object. Then, it creates an email message with the specified subject and HTML content. It sets the email sender name and address from the `settings` object. It then defines the SMTP options such as host, port, TLS or SSL, user, and password from the `settings` object. Finally, the function sends the email using the defined settings and logs the result to a file called `logging`. The function returns None. Overall, this function is responsible for sending an email with the specified parameters using the SMTP protocol.

```
The user input is a Python function that sends an email using the `emails` module and the `logging` module to log the response from the email sending API.

The system parses the function to identify its purpose, parameters, and
```

return value, and then generates a commented version of the function based on this analysis. The following is the commented code:
```
""" Function to send an email using the `emails` module and logging module"""
def send_email(
""" Recipient email address"""
*,
email_to: str,
""" Subject of the email"""
subject: str = "",
""" HTML content of the email"""
html_content: str = "",
):
""" Check if email sending is enabled in the configuration"""
assert settings.emails_enabled, "no provided configuration for email variables"

""" Create a message object with the specified subject and HTML content"""
message = emails.Message(
subject=subject,
html=html_content,
mail_from=(settings.EMAILS_FROM_NAME, settings.EMAILS_FROM_EMAIL),
)

""" Set the SMTP options for sending the email"""
smtp_options = {"host": settings.SMTP_HOST, "port": settings.SMTP_PORT}
if settings.SMTP_TLS:
smtp_options["tls"] = True
elif settings.SMTP_SSL:
smtp_options["ssl"] = True

""" Set the email sending user and password if provided in the configuration"""
if settings.SMTP_USER:
smtp_options["user"] = settings.SMTP_USER
if settings.SMTP_PASSWORD:
smtp_options["password"] = settings.SMTP_PASSWORD

""" Send the email using the message object and SMTP options"""
response = message.send(to=email_to, smtp=smtp_options)

""" Log the result of the email sending"""
logging.info(f"send email result: {response}")
```
In this commented code, each line is preceded by a `#` symbol and followed by an explanation of what the line does. The lines that set the SMTP options for sending the email are condensed into one line using Python's `if` statement to make it easier to read. The logging statement is also commented out, as the user may not want to log the result of the email sending.

verify_password_reset_token

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/utils.py

This Python function `verify_password_reset_token` takes a string argument `token` and returns either a string or `None`. It first tries to decode the provided token using the `jwt.decode()` method with the secret key and algorithm specified in the `settings.SECRET_KEY` and `algorithms` variables, respectively. If the decoding is successful, it extracts the value of the 'sub' (subject) field from the decoded object and returns it as a string. Otherwise, it returns `None`. This function is likely part of a password reset functionality, where a token is generated and sent to the user via email or SMS, which can then be used to reset their password. The purpose of this function is to verify the validity and authenticity of these tokens before allowing the password reset operation.

```
```
""" Verify a password reset token"""
"""""""
""" Purpose:"""
""" * Decodes a password reset token and returns the associated user
ID."""
"""""""
""" Parameters:"""
""" * token: The password reset token to be verified."""
"""""""
""" Return value:"""
""" * str or None: If the token is valid, returns the user ID associated
with it. Otherwise, returns None."""
def verify_password_reset_token(token: str) -> str | None:
try:
""" Attempt to decode the password reset token using the secret key and
HS256 algorithm"""
decoded_token = jwt.decode(token, settings.SECRET_KEY,
algorithms=["HS256"])

""" If the token is valid, return the user ID associated with it"""
return str(decoded_token["sub"])
except JWTError:
""" If the token is invalid, return None"""
return None
```
```

get_user_by_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/crud.py

This Python function, named `get_user_by_email`, takes two arguments: a SQLAlchemy session (`session`) and an email address (`email`). It retrieves a user from the database by their email using the SQLAlchemy ORM (Object-Relational Mapping) and returns the user object (`User`) stored in the session variable or `None` if the user with the provided email does not exist. The function is annotated with type hints, indicating that `session` should be a specific type of session (`Session`) and `email` should be a string (`str`).

Here is the commented version of the function based on the analysis:
```
def get_user_by_email(*, session: Session, email: str) -> User | None:
"""
This function retrieves a user from the database using their email
address.

Parameters:
* session (Session): The SQLAlchemy session object used to interact with
the database.
* email (str): The email address of the user to retrieve.

Returns:
A User object if the email address is found in the database, or None if
no such user exists.
"""
statement = select(User).where(User.email == email)
session_user = session.exec(statement).first()
return session_user
```

get_password_hash

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/core/security.py

This Python function, named `get_password_hash`, takes a string `password` as its argument and returns the hash of that password using the `pwd_context` object for password hashing. The function's purpose is to securely store passwords in a database or other storage mechanism by converting them into a fixed-size value, which can be quickly compared without exposing the actual password. This helps prevent unauthorized access and protect user data.

```
[PYTHON]
def get_password_hash(password: str) -> str:
"""
Generates a password hash using the given password string.

Args:
password (str): The password to be hashed.

Returns:
str: The generated password hash.
"""
return pwd_context.hash(password)
[/PYTHON]
[TESTS]
""" Test case 1:"""
assert get_password_hash("hello") == "some_hashed_string"

""" Test case 2:"""
```

```
assert get_password_hash("world") == "some_other_hashed_string"
[/TESTS]
```

get_user_by_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/crud.py

The Python function `get_user_by_email` takes in a SQLAlchemy session and an email as arguments, and returns the corresponding user object from the session or None if not found. It achieves this by executing a SELECT query with a WHERE clause to filter for the specific email, and returning the resulting first row from the query. This function is likely part of a larger application that uses SQLAlchemy as an ORM (Object-Relational Mapping) tool to interact with a relational database. The * parameter after the function name is a Python syntax called "function annotations", which allows the function to take optional arguments with default values, but also supports keyword arguments. In this case, it's used to document the expected type of the session argument.

```
[PYTHON]
def get_user_by_email(*, session: Session, email: str) -> User | None:
"""
This function retrieves a user from the database using their email
address.

Args:
session: A SQLAlchemy session object.
email: The email address of the user to retrieve.

Returns:
A User object if a user with the specified email address exists, or None
if no such user was found.
"""
statement = select(User).where(User.email == email)
session_user = session.exec(statement).first()
return session_user
[/PYTHON]
```

generate_password_reset_token

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/utils.py

This Python function named `generate_password_reset_token` takes an email address as input and returns a string representing a password reset token. The function first calculates the number of hours before the token expires based on a setting called `EMAIL_RESET_TOKEN_EXPIRE_HOURS`. It then gets the current UTC time, adds the calculated expiry duration to it, and converts the resulting datetime object to an integer representing the Unix timestamp. The function uses the Python library `jwt` (JSON Web Tokens) to encode a JSON payload containing the token's expiration time, creation time, and the email address provided as input. The encoded string is returned as the password reset

token. In summary, this function generates a secure and time-bound password reset token for the given email address using `jwt`.

```
```
""" Generate a password reset token for the given email address"""
def generate_password_reset_token(email: str) -> str:
""" Create a datetime object representing the current time in UTC"""
now = datetime.utcnow()

""" Calculate the expiration date of the token by adding the specified
number of hours to the current time"""
delta = timedelta(hours=settings.EMAIL_RESET_TOKEN_EXPIRE_HOURS)
expires = now + delta

""" Convert the expiration date to a Unix timestamp for use in the JWT"""
exp = expires.timestamp()

""" Encode the JWT using the secret key and HS256 algorithm"""
encoded_jwt = jwt.encode(
{"exp": exp, "nbf": now, "sub": email},
settings.SECRET_KEY,
algorithm="HS256",
)

""" Return the encoded JWT as a string"""
return encoded_jwt
```
```

generate_reset_password_email

/Users/williambrach/Developer/hackkosice/hk-2024/backend-hk24/app/uploads/repos/full-stack-fastapi-template/backend/app/utils.py

The Python function `generate_reset_password_email` takes in three arguments: `email_to`, which is the recipient's email address; `email`, which is the user's email address whose password is being reset; and `token`, which is a unique string for verifying the reset request. The function returns an object of type `EmailData`, which contains both HTML content and subject for the email that will be sent to the recipient. The email template is generated using Django's `render_email_template` function, which takes in a template name (in this case, "reset_password.html") and passes context data such as the project name, user's email and username, reset token's validity duration, and the link to the password reset page. Overall, this function generates an email containing instructions for resetting the password of a particular user account using a unique token provided in the email.

```
Here is the commented version of the function:
```
def generate_reset_password_email(email_to: str, email: str, token: str)
-> EmailData:
"""
Generate an email to allow the user to reset their password.

Args:
```
```

```
    email_to (str): The recipient's email address.
    email (str): The user's email address.
    token (str): A security token used to validate the request.

    Returns:
    EmailData: An object containing the email content and subject line.
    """
    project_name = settings.PROJECT_NAME
    subject = f"{project_name} - Password recovery for user {email}"
    link = f"{settings.server_host}/reset-password?token={token}"
    html_content = render_email_template(
    template_name="reset_password.html",
    context={
    "project_name": settings.PROJECT_NAME,
    "username": email,
    "email": email_to,
    "valid_hours": settings.EMAIL_RESET_TOKEN_EXPIRE_HOURS,
    "link": link,
    },
    )
    return EmailData(html_content=html_content, subject=subject)
```

Project structure

|--- app
| |--- alembic
| |--- api
| |--- core
| |--- email-templates
| |---- tests
| |----__init__.py
| |----backend_pre_start.py
| |----crud.py
| |----authenticate()
| |----create_user()
| |----get_user_by_email()
| |----initial_data.py
| |----main.py
| |----models.py
| |----tests_pre_start.py
| |----utils.py
| |----generate_new_account_email()
| |----generate_password_reset_token()
| |----generate_reset_password_email()
| |----generate_test_email()
| |----send_email()
| |----verify_password_reset_token()
|--- app/alembic
| |---- versions
| |----env.py

```
|--- app/alembic/versions
| |----e2412789c190_initialize_models.py
|--- app/api
| |---- routes
| |----__init__.py
| |----deps.py
| |----main.py
|--- app/api/routes
| |----__init__.py
| |----items.py
| |----login.py
| |----users.py
| |----create_user()
| |----utils.py
|--- app/core
| |----__init__.py
| |----config.py
| |----db.py
| |----security.py
| |----create_access_token()
| |----get_password_hash()
| |----verify_password()
|--- app/email-templates
| |--- build
| |---- src
|--- app/email-templates/build
|--- app/email-templates/src
|--- app/tests
| |--- api
| |--- crud
| |--- scripts
| |---- utils
| |----__init__.py
| |----conftest.py
|--- app/tests/api
| |---- routes
| |----__init__.py
|--- app/tests/api/routes
| |----__init__.py
| |----test_items.py
| |----test_login.py
| |----test_users.py
|--- app/tests/crud
| |----__init__.py
| |----test_user.py
|--- app/tests/scripts
| |----__init__.py
| |----test_backend_pre_start.py
| |----test_test_pre_start.py
|--- app/tests/utils
| |----__init__.py
| |----item.py
| |----user.py
```

| |----utils.py
|--- root
| |--- app
| |---- scripts
|---- scripts