my-example-refactor

Readme summary

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

The project is configured 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 and run the migration in the database. Alembic is already configured to import your SQLModel models from `./backend/app/models.py`.

Contribution

The project allows for contributions. However, contributors are expected to follow the outlined workflow and use the provided configurations for local development, testing, and migrations.

Business Stories

users.py

Route: /
Functions:
- `get_user_by_email`
- `create_user`
- `generate_new_account_email`
- `send_email`
- `get_password_hash`

Route: /me
Functions: No functions

Route: /signup
Functions:
- `get_user_by_email`
- `create_user`
- `get_password_hash`

Route: /{user_id}
Functions: No functions

utils.py

Route: /test-email/
Functions:
- `generate_test_email`
- `send_email`

login.py

Route: /login/access-token
Functions:
- `authenticate`
- `create_access_token`
- `get_user_by_email`
- `verify_password`

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`

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

items.py

Route: /
Functions: No functions

Route: /{id}
Functions: No functions

Class Data

Based on the provided class definitions, it appears that this project involves a user authentication and management system, as well as a functionality for creating, updating, and deleting items. The `UserBase` class represents the common attributes of all users, while `UserCreate`, `UserRegister`, `UserUpdateMe`, and `UserUpdate` define the parameters required for creating new users, registering new users, updating existing user information (excluding password), and updating email and full name respectively.

The `UpdatePassword` class allows users to update their current password before setting a new one. The `UserOut` and `UsersOut` classes are used to return the details of individual users or lists of users respectively, while the `ItemBase` and `ItemCreate` classes define the basic attributes of items and parameters required for creating new items respectively.

The `ItemUpdate` class allows existing item information to be updated, and `ItemOut` and `ItemsOut` return individual item details or lists of items respectively. The `Message` class seems to represent messages used in communication between users.

The `Token`, `TokenPayload`, and `NewPassword` classes relate to token management and password reset functionality. There are also several settings-related classes, including `EmailData`, which appears to be related to email communications, as well as various server configuration settings such as `SECRET_KEY`, `ACCESS_TOKEN_EXPIRE_MINUTES`, etc.

Overall, these class definitions seem to cover a wide range of functionalities and data structures required for this web application.

Class Data Comments

UserBase

Here's an example of how the commented function might look like based on the given user input:

```
def get_user_details(user):
"""
This function takes a user object as input and returns a tuple containing their email address,
boolean indicating whether they are actively using the system, boolean indicating whether
they have administrative privileges, and their full name. The inputs are:
- 'email': user's unique email address used for login
- 'is_active': True if the user is currently using the system, False otherwise
- 'is_superuser': True if the user has administrator privileges, False otherwise
- 'full_name': The full name of the user as entered during registration
"""
return user.email, user.is_active, user.is_superuser, user.full_name
```

UserCreate

```
Def generate_commented_function(password):
"""
This function takes a password as input and returns None (as the password is not returned for security purposes). The input password is processed and stored securely in the system. It does not perform any additional functionality beyond this. Note that for optimal security, it is recommended to use a dedicated password management library or framework instead of implementing your own password storage logic.
:param password: A string containing the user's password.
"""
""" Securely store the password in the system using industry-standard encryption techniques and access control mechanisms."""
pass # Returning None to prevent accidental exposure of the password.
```

UserRegister

```
Def get_user_information(email, password, full_name):
"""
This function takes in three arguments representing a user's email address, password, and full name.
```

```
It does not return anything but rather collects this information for use
elsewhere in the program.
:param email: The unique identifier for the user's account.
:param password: A secret string used to verify the user's identity.
:param full_name: The user's given name and surname, displayed publicly.
"""
```

The comments provide context and clarity around the function's parameters and purpose, making it easier for other developers (or future versions of yourself) to understand and use the function.

UserUpdate

Here's the commented version of that function:

```
def get_credentials(email: str, password: str) -> tuple:
"""
This function returns a tuple containing the user's email and password.

:param email: The email address of the user. Type: str.
:param password: The password of the user. Type: str.
:return: A tuple containing the email and password. Type: tuple.
"""
```

In this example, the function takes two arguments `email` (type `str`) and `password` (also type `str`). It returns a tuple containing both these values. The comments above the function header describe the function's purpose, parameters, and return value.

UserUpdateMe

```
Def get_user_info(required_fields):
"""
This function takes a list of required fields as input and returns a
dictionary containing the necessary user information for those fields.

Args:
required_fields (list): A list of field names that are required for the
user info.

Returns:
dict: A dictionary containing the user information for the specified
fields.
"""
```

UpdatePassword

Here's the commented version of this Python function:

```
def change_password(current_password, new_password):
"""
This function takes two passwords as arguments - the current password and
the new password.
The purpose of this function is to change the user's password.

Arguments:
current_password (str): The user's current password.
new_password (str): The new password that the user wants to set.

Returns:
None, but the function updates the user's password in the database.
"""
```

I hope this helps clarify the purpose and functionality of the function for you! Let me know if there's anything else I can do for you.

User

```
Def get_user_data(user_info: dict) -> list[str]:
"""
This function takes a dictionary representing user information,
and returns a list of the values for the keys 'id', 'hashed_password',
and 'items'.

Args:
- user_info (dict): A dictionary containing user data. The keys are
expected to be 'id', 'hashed_password', and 'items'.
- Return type (list[str]): The function returns a list of strings,
containing the values for the specified keys.
"""
```

UserOut

```
""" Define a function called `get_id` that takes no arguments (since the
input is a list with only one item, an empty tuple would also work here)
and returns that single item"""
def get_id():
""" The `['id']` input represents a list with a single string 'id'. We
return this string."""
return 'id'
```

Note: If the user input was something more complex, such as:

```
def calculate_average(list1, list2):
total = 0
for num in list1 + list2:
total += num
average = total / (len(list1) + len(list2))
return average
```

The commented version would look like this:

```
""" Define a function called `calculate_average` that takes two lists,
`list1` and `list2`, as arguments"""
""" We initialize a variable `total` to 0, then loop through both input
lists (`list1` and `list2`) using the `for num in list1 + list2:`
syntax."""
""" Inside the loop, we add each number (`num`) to the `total`
variable."""
""" After all numbers have been added, we calculate the average by
dividing the total by the sum of the lengths of both input lists."""
def calculate_average(list1, list2):
total = 0
for num in list1 + list2:
total += num
average = total / (len(list1) + len(list2))
return average
```

UsersOut

Here's how the system could generate a commented version of this function based on its analysis:

```
""" This function takes in two parameters, 'data' and 'count'."""
""" It does not return anything."""
def my_function(data, count):
""" We iterate over the 'data' list 'count' number of times."""
for I in range(count):
""" For each iteration, we do something with the current element of
'data'."""
""" In this example, we simply print it."""
print(data[i])
```

ItemBase

```
Def get_metadata(title, description):
"""
This function takes in two arguments, `title` and `description`, and
returns a tuple containing both of them as metadata.
:param title: A string representing the title of the item being
described.
:param description: A string representing a detailed description of the
item being described.
:return: A tuple containing both the title and description as metadata.
"""
```

ItemCreate

```
""" This function takes a single argument 'title' (a string) and returns
it as is"""
def title(title: str) -> str:
""" No logic implemented inside the function, simply passing the argument
back"""
return title
```

ItemUpdate

```
""" This function takes in a single parameter, 'title', and returns only
that value"""
def return_title(title):
"""
This function accepts a single argument, 'title', which is expected to be
a string.
The function simply returns the input title without any modifications.
"""
return title
```

Item

```
Def get_column_names(collection):
"""
This function returns a list containing the names of all columns in the
given MongoDB collection.

Parameters:
- collection (pymongo.Collection): A MongoDB collection object to fetch
column names from.

Returns:
- list: A list containing the names of all columns in the collection.
```

```
"""
return ['id', 'title', 'owner_id', 'owner'] # This is the original
function body with comments explaining its purpose, parameters, and
return value.
```

ItemOut

Here's how the system would generate commented code for this user input:

```
""" Function to extract 'id' and 'owner_id' fields from a dictionary-like
object"""
""" Parameters:"""
""" - dictionary (dict): The input object containing the desired
fields"""
""" Returns:"""
""" - tuple of values for 'id' and 'owner_id', or None if not found"""
def extract_fields(dictionary):
""" Check if 'id' is present in the dictionary"""
id = dictionary.get('id')
""" If 'id' is not present, return None for that field"""
if id is None:
return None, dictionary.get('owner_id')
""" Otherwise, extract both fields and return them as a tuple"""
owner_id = dictionary.get('owner_id')
return id, owner_id
```

Note: The exact behavior of the function may differ based on the input format and desired output type, but this example should provide a general idea of how the system would generate commented code for this specific user input.

ItemsOut

```
def count_occurrences(data, count=None):
"""
This function takes a list of data as input and returns the number of
occurrences of a specific item in it.
If no item is specified (i.e., the `count` parameter is not provided),
the function returns a dictionary containing the number of occurrences of
each unique item in the data.

Args:
- data: A list or tuple containing the input data.
- count (optional): The specific item to count. If not provided, all
items will be counted. Default is None.
Returns:
```

```
- int: The number of occurrences of the specified `count` item in the
data, if provided. Otherwise, a dictionary containing the counts for all
unique items in the data.
"""
```

Message

```
```
""" This function returns a message"""
def message():
pass
```

Or for a more complex example, let's say the user inputs:
```
['calculate_area', 'Computes the area of a rectangle based on its width
and height']
```

Then the commented version would be:
```
""" Function to calculate the area of a rectangle based on its width and
height"""
def calculate_area(width, height):
""" Calculate the area using the formula width * height"""
area = width * height
return area
```

## Token

Here's an example of how the system might generate commented code for this input function:

```
def get_authentication_tokens(access_token: str, token_type: str) ->
tuple[str, str]:
"""
Retrieves a user's authentication tokens from the authentication server.

Parameters:
access_token (str): The authorization grant issued to the client by the
resource server.
token_type (str): The type of token returned, such as "bearer" or "mac".

Returns:
tuple[str, str]: A tuple containing the user's access token and token
type.
"""
```
```

```
""" Code to retrieve authentication tokens goes here..."""
```

Note that in this example, the system has identified that the function takes two positional arguments (`access_token` and `token_type`) of types `str`, returns a tuple containing both `str` values as its return value. It also accurately identifies what the function does and how it is used in its docstring.

TokenPayload

```
Def sub(x: float, y: float) -> float:
"""
This function takes two floating point numbers, x and y, as input and
returns their difference, which is calculated by subtracting y from x.

Parameters:
x (float): The first number.
y (float): The second number.

Returns:
float: The difference between x and y.
"""
```

In this case, the user input is just the name of the function, `sub`. Based on that, we can deduce the function's purpose, parameters, and return type using Python's syntax rules.

The function `sub` takes two arguments `x` and `y`, both of which are expected to be float types. It returns a float value representing their difference. The function is named `sub`, which suggests that it performs subtraction operation between the input parameters.

Using this information, we can create a commented version of the function. Note that while generating comments for the function, we assume that the user has provided the correct syntax and input arguments. In case of errors in the input code or unexpected inputs to the function, the comments might not be accurate.

NewPassword

Here's the commented version of the function based on its signature:

```
""" Function to reset user's password"""
""" Takes two parameters, 'token' (a string representing the reset link)
and 'new_password' (a new password as a string)"""
def reset_password(token: str, new_password: str) -> None:
""" The function returns None since password resets are not expected to
return anything useful."""
```

```

The comments provide clarity about the function's purpose and parameters, making it easier for others (or future self) to understand and maintain the code.

## EmailData

```
def send_email(html_content: str, subject: str) -> None:
"""
Sends an email with the given HTML content and subject.

:param html_content: The HTML content of the email message.
:param subject: The subject line of the email.
:return: None (as this is a void function).
"""
```

Explanation:

- The first line of the commented version contains the original function name and its parameters in parentheses, with their respective types. This helps to clarify what arguments are expected by the function.
- The docstring (a string enclosed in triple quotes) provides a detailed description of what the function does, including any input parameters, output values, and potential exceptions or error conditions.
- The return type is also specified, which is useful for functions that return a value, as it helps to ensure that the correct data type is returned. In this case, since `send_email()` is a void function (meaning it does not return any value), we set the return type to `None`.
- The docstring should include a detailed description of what the function does, how it works, and any important caveats or limitations that users should be aware of.
- In general, comments should be added in a way that enhances clarity and readability, without being overly verbose or redundant.

## Settings

Here's a commented version of the function based on the analysis:

```
def configure_app(model_config: str, # Config for model server
API_V1_STR: str = 'v1', # Default version is v1
SECRET_KEY: str = None, # Required for session management
ACCESS_TOKEN_EXPIRE_MINUTES: int = 60 * 5, # Default access token
expiration time
DOMAIN: str = 'localhost', # Domain for the app
ENVIRONMENT: str = 'development', # Development or production environment
BACKEND_CORS_ORIGINS: List[str] = ['http://localhost:3000',
```
```

```
'https://example.com'], # Allowed cross-origin resource sharing (CORS)
origins
PROJECT_NAME: str = None, # Project name for logging and error reporting
SENTRY_DSN: str = None, # Optional Sentry error tracking integration
POSTGRES_SERVER: str = 'localhost', # PostgreSQL server hostname or IP
address
POSTGRES_PORT: int = 5432, # Default PostgreSQL port number
POSTGRES_USER: str = None, # Username for accessing the database
POSTGRES_PASSWORD: str = None, # Password for accessing the database
POSTGRES_DB: str = None, # Database name to use
SMTP_TLS: bool = False, # Enable or disable STARTLS for sending emails
SMTP_SSL: bool = False, # Enable or disable SSL/TLS for sending emails
SMTP_PORT: int = 587, # Default SMTP port number
SMTP_HOST: str = 'smtp.gmail.com', # Gmail SMTP server hostname or IP
address
SMTP_USER: str = None, # Email address to use for sending emails
SMTP_PASSWORD: str = None, # Password for accessing the email account
EMAILS_FROM_EMAIL: str = 'noreply@example.com', # Email address to send
from
EMAILS_FROM_NAME: str = 'Example App', # Name to use when sending emails
EMAIL_RESET_TOKEN_EXPIRE_HOURS: int = 24, # Number of hours before reset
token expires
EMAIL_TEST_USER: bool = False, # Enable or disable testing user
functionality
FIRST_SUPERUSER: str = None, # First superuser username (for testing and
development)
FIRST_SUPERUSER_PASSWORD: str = None, # Password for the first superuser
account
USERS_OPEN_REGISTRATION: bool = True, # Enable or disable user
registration functionality
):
"""
Configure various settings and options for the app.

:param model_config: Config for the model server (e.g., address, port
number).
:type model_config: str

:param API_V1_STR: Default version of the API (e.g., 'v1').
:type API_V1_STR: str

:param SECRET_KEY: Required for session management.
:type SECRET_KEY: str

:param ACCESS_TOKEN_EXPIRE_MINUTES: Default access token expiration time
(in minutes).
:type ACCESS_TOKEN_EXPIRE_MINUTES: int

:param DOMAIN: Domain for the app.
:type DOMAIN: str

:param ENVIRONMENT: Development or production environment (e.g.,
'development' or 'production').
```

```
:type ENVIRONMENT: str

:param BACKEND_CORS_ORIGINS: List of allowed cross-origin resource
sharing (CORS) origins (e.g., ['http://localhost:3000',
'https://example.com']).
:type BACKEND_CORS_ORIGINS: List[str]

:param PROJECT_NAME: Project name for logging and error reporting
(optional).
:type PROJECT_NAME: str

:param SENTRY_DSN: Optional Sentry error tracking integration DSN (URL or
secret key).
:type SENTRY_DSN: str

:param POSTGRES_SERVER: PostgreSQL server hostname or IP address.
:type POSTGRES_SERVER: str

:param POSTGRES_PORT: Default PostgreSQL port number (5432).
:type POSTGRES_PORT: int

:param POSTGRES_USER: Username for accessing the database.
:type POSTGRES_USER: str

:param POSTGRES_PASSWORD: Password for accessing the database.
:type POSTGRES_PASSWORD: str

:param POSTGRES_DB: Database name to use (optional).
:type POSTGRES_DB: str

:param SMTP_TLS: Enable or disable STARTLS for sending emails.
:type SMTP_TLS: bool

:param SMTP_SSL: Enable or disable SSL/TLS for sending emails.
:type SMTP_SSL: bool

:param SMTP_PORT: Default SMTP port number (587).
:type SMTP_PORT: int

:param SMTP_HOST: Gmail SMTP server hostname or IP address.
:type SMTP_HOST: str

:param SMTP_USER: Email address to use for sending emails (optional).
:type SMTP_USER: str

:param SMTP_PASSWORD: Password for accessing the email account
(optional).
:type SMTP_PASSWORD: str

:param EMAILS_FROM_EMAIL: Email address to send from.
:type EMAILS_FROM_EMAIL: str
```

```
:param EMAILS_FROM_NAME: Name to use when sending emails (optional).
:type EMAILS_FROM_NAME: str

:param EMAIL_RESET_TOKEN_EXPIRE_HOURS: Number of hours before reset token
expires.
:type EMAIL_RESET_TOKEN_EXPIRE_HOURS: int

:param EMAIL_TEST_USER: Enable or disable testing user functionality
(optional).
:type EMAIL_TEST_USER: bool

:param FIRST_SUPERUSER: First superuser username (for testing and
development) (optional).
:type FIRST_SUPERUSER: str

:param FIRST_SUPERUSER_PASSWORD: Password for the first superuser account
(optional).
:type FIRST_SUPERUSER_PASSWORD: str

:param USERS_OPEN_REGISTRATION: Enable or disable user registration
functionality (optional).
:type USERS_OPEN_REGISTRATION: bool
"""
```

Function to Test

Session

```
def db() -> Generator[Session, None, None]:
with Session(engine) as session:
init_db(session)
yield session
statement = delete(Item)
session.execute(statement)
statement = delete(User)
session.execute(statement)
session.commit()
```

init_db

```
def db() -> Generator[Session, None, None]:
with Session(engine) as session:
init_db(session)
yield session
statement = delete(Item)
session.execute(statement)
statement = delete(User)
session.execute(statement)
session.commit()
```

delete

```
def db() -> Generator[Session, None, None]:
with Session(engine) as session:
init_db(session)
yield session
statement = delete(Item)
session.execute(statement)
statement = delete(User)
session.execute(statement)
session.commit()
```

session.execute

```
def db() -> Generator[Session, None, None]:
with Session(engine) as session:
init_db(session)
yield session
statement = delete(Item)
session.execute(statement)
statement = delete(User)
session.execute(statement)
session.commit()
```

session.commit

```
def db() -> Generator[Session, None, None]:
with Session(engine) as session:
init_db(session)
yield session
statement = delete(Item)
session.execute(statement)
statement = delete(User)
session.execute(statement)
session.commit()
```

pytest.fixture

```
def normal_user_token_headers(client: TestClient, db: Session) ->
dict[str, str]:
return authentication_token_from_email(
client=client, email=settings.EMAIL_TEST_USER, db=db
)
```

TestClient

```
def client() -> Generator[TestClient, None, None]:
with TestClient(app) as c:
yield c
```

get_superuser_token_headers

```
def superuser_token_headers(client: TestClient) -> dict[str, str]:
return get_superuser_token_headers(client)
```

authentication_token_from_email

```
def normal_user_token_headers(client: TestClient, db: Session) ->
dict[str, str]:
return authentication_token_from_email(
client=client, email=settings.EMAIL_TEST_USER, db=db
)
```

client.post

```
def user_authentication_headers(
*, client: TestClient, email: str, password: str
) -> dict[str, str]:
data = {"username": email, "password": password}
r = client.post(f"{settings.API_V1_STR}/login/access-token", data=data)
response = r.json()
auth_token = response["access_token"]
headers = {"Authorization": f"Bearer {auth_token}"}
return headers
```

r.json

```
def user_authentication_headers(
*, client: TestClient, email: str, password: str
```

```
) -> dict[str, str]:
data = {"username": email, "password": password}
r = client.post(f"{settings.API_V1_STR}/login/access-token", data=data)
response = r.json()
auth_token = response["access_token"]
headers = {"Authorization": f"Bearer {auth_token}"}
return headers
```

random_email

```
def create_random_user(db: Session) -> User:
email = random_email()
password = random_lower_string()
user_in = UserCreate(email=email, password=password)
user = crud.create_user(session=db, user_create=user_in)
return user
```

random_lower_string

```
def authentication_token_from_email(
*, client: TestClient, email: str, db: Session
) -> dict[str, str]:
"""
Return a valid token for the user with given email.
If the user doesn't exist it is created first.
"""
password = random_lower_string()
user = crud.get_user_by_email(session=db, email=email)
if not user:
user_in_create = UserCreate(email=email, password=password)
user = crud.create_user(session=db, user_create=user_in_create)
else:
user_in_update = UserUpdate(password=password)
if not user.id:
raise Exception("User id not set")
user = crud.update_user(session=db, db_user=user,
user_in=user_in_update)
return user_authentication_headers(client=client, email=email,
password=password)
```

UserCreate

```
def authentication_token_from_email(
*, client: TestClient, email: str, db: Session
) -> dict[str, str]:
"""
Return a valid token for the user with given email.
If the user doesn't exist it is created first.
"""
password = random_lower_string()
user = crud.get_user_by_email(session=db, email=email)
if not user:
user_in_create = UserCreate(email=email, password=password)
```

```
user = crud.create_user(session=db, user_create=user_in_create)
else:
user_in_update = UserUpdate(password=password)
if not user.id:
raise Exception("User id not set")
user = crud.update_user(session=db, db_user=user,
user_in=user_in_update)
return user_authentication_headers(client=client, email=email,
password=password)
```

crud.create_user

```
def authentication_token_from_email(
*, client: TestClient, email: str, db: Session
) -> dict[str, str]:
"""
Return a valid token for the user with given email.
If the user doesn't exist it is created first.
"""
password = random_lower_string()
user = crud.get_user_by_email(session=db, email=email)
if not user:
user_in_create = UserCreate(email=email, password=password)
user = crud.create_user(session=db, user_create=user_in_create)
else:
user_in_update = UserUpdate(password=password)
if not user.id:
raise Exception("User id not set")
user = crud.update_user(session=db, db_user=user,
user_in=user_in_update)
return user_authentication_headers(client=client, email=email,
password=password)
```

crud.get_user_by_email

```
def authentication_token_from_email(
*, client: TestClient, email: str, db: Session
) -> dict[str, str]:
"""
Return a valid token for the user with given email.
If the user doesn't exist it is created first.
"""
password = random_lower_string()
user = crud.get_user_by_email(session=db, email=email)
if not user:
user_in_create = UserCreate(email=email, password=password)
user = crud.create_user(session=db, user_create=user_in_create)
else:
user_in_update = UserUpdate(password=password)
if not user.id:
raise Exception("User id not set")
user = crud.update_user(session=db, db_user=user,
user_in=user_in_update)
```

```
return user_authentication_headers(client=client, email=email,
password=password)
```

UserUpdate

```
def authentication_token_from_email(
*, client: TestClient, email: str, db: Session
) -> dict[str, str]:
"""
Return a valid token for the user with given email.
If the user doesn't exist it is created first.
"""
password = random_lower_string()
user = crud.get_user_by_email(session=db, email=email)
if not user:
user_in_create = UserCreate(email=email, password=password)
user = crud.create_user(session=db, user_create=user_in_create)
else:
user_in_update = UserUpdate(password=password)
if not user.id:
raise Exception("User id not set")
user = crud.update_user(session=db, db_user=user,
user_in=user_in_update)
return user_authentication_headers(client=client, email=email,
password=password)
```

Exception

```
def authentication_token_from_email(
*, client: TestClient, email: str, db: Session
) -> dict[str, str]:
"""
Return a valid token for the user with given email.
If the user doesn't exist it is created first.
"""
password = random_lower_string()
user = crud.get_user_by_email(session=db, email=email)
if not user:
user_in_create = UserCreate(email=email, password=password)
user = crud.create_user(session=db, user_create=user_in_create)
else:
user_in_update = UserUpdate(password=password)
if not user.id:
raise Exception("User id not set")
user = crud.update_user(session=db, db_user=user,
user_in=user_in_update)
return user_authentication_headers(client=client, email=email,
password=password)
```

crud.update_user

```
def authentication_token_from_email(
*, client: TestClient, email: str, db: Session
```

```
) -> dict[str, str]:
"""
Return a valid token for the user with given email.
If the user doesn't exist it is created first.
"""
password = random_lower_string()
user = crud.get_user_by_email(session=db, email=email)
if not user:
user_in_create = UserCreate(email=email, password=password)
user = crud.create_user(session=db, user_create=user_in_create)
else:
user_in_update = UserUpdate(password=password)
if not user.id:
raise Exception("User id not set")
user = crud.update_user(session=db, db_user=user,
user_in=user_in_update)
return user_authentication_headers(client=client, email=email,
password=password)
```

user_authentication_headers

```
def authentication_token_from_email(
*, client: TestClient, email: str, db: Session
) -> dict[str, str]:
"""
Return a valid token for the user with given email.
If the user doesn't exist it is created first.
"""
password = random_lower_string()
user = crud.get_user_by_email(session=db, email=email)
if not user:
user_in_create = UserCreate(email=email, password=password)
user = crud.create_user(session=db, user_create=user_in_create)
else:
user_in_update = UserUpdate(password=password)
if not user.id:
raise Exception("User id not set")
user = crud.update_user(session=db, db_user=user,
user_in=user_in_update)
return user_authentication_headers(client=client, email=email,
password=password)
```

random.choices

```
def random_lower_string() -> str:
return "".join(random.choices(string.ascii_lowercase, k=32))
```

create_random_user

```
def create_random_item(db: Session) -> Item:
user = create_random_user(db)
owner_id = user.id
assert owner_id is not None
```

```
title = random_lower_string()
description = random_lower_string()
item_in = ItemCreate(title=title, description=description)
return crud.create_item(session=db, item_in=item_in, owner_id=owner_id)
```

ItemCreate

```
def create_random_item(db: Session) -> Item:
user = create_random_user(db)
owner_id = user.id
assert owner_id is not None
title = random_lower_string()
description = random_lower_string()
item_in = ItemCreate(title=title, description=description)
return crud.create_item(session=db, item_in=item_in, owner_id=owner_id)
```

crud.create_item

```
def create_random_item(db: Session) -> Item:
user = create_random_user(db)
owner_id = user.id
assert owner_id is not None
title = random_lower_string()
description = random_lower_string()
item_in = ItemCreate(title=title, description=description)
return crud.create_item(session=db, item_in=item_in, owner_id=owner_id)
```

MagicMock

```
def test_init_successful_connection() -> None:
engine_mock = MagicMock()
session_mock = MagicMock()
exec_mock = MagicMock(return_value=True)
session_mock.configure_mock(**{"exec.return_value": exec_mock})
with (
patch("sqlmodel.Session", return_value=session_mock),
patch.object(logger, "info"),
patch.object(logger, "error"),
patch.object(logger, "warn"),
):
try:
init(engine_mock)
connection_successful = True
except Exception:
connection_successful = False
assert (
connection_successful
), "The database connection should be successful and not raise an
exception."
assert session_mock.exec.called_once_with(
select(1)
), "The session should execute a select statement once."
```

session_mock.configure_mock

```
def test_init_successful_connection() -> None:
engine_mock = MagicMock()
session_mock = MagicMock()
exec_mock = MagicMock(return_value=True)
session_mock.configure_mock(**{"exec.return_value": exec_mock})
with (
patch("sqlmodel.Session", return_value=session_mock),
patch.object(logger, "info"),
patch.object(logger, "error"),
patch.object(logger, "warn"),
):
try:
init(engine_mock)
connection_successful = True
except Exception:
connection_successful = False
assert (
connection_successful
), "The database connection should be successful and not raise an
exception."
assert session_mock.exec.called_once_with(
select(1)
), "The session should execute a select statement once."
```

patch

```
def test_init_successful_connection() -> None:
engine_mock = MagicMock()
session_mock = MagicMock()
exec_mock = MagicMock(return_value=True)
session_mock.configure_mock(**{"exec.return_value": exec_mock})
with (
patch("sqlmodel.Session", return_value=session_mock),
patch.object(logger, "info"),
patch.object(logger, "error"),
patch.object(logger, "warn"),
):
try:
init(engine_mock)
connection_successful = True
except Exception:
connection_successful = False
assert (
connection_successful
), "The database connection should be successful and not raise an
exception."
assert session_mock.exec.called_once_with(
select(1)
), "The session should execute a select statement once."
```

patch.object

```
def test_init_successful_connection() -> None:
engine_mock = MagicMock()
session_mock = MagicMock()
exec_mock = MagicMock(return_value=True)
session_mock.configure_mock(**{"exec.return_value": exec_mock})
with (
patch("sqlmodel.Session", return_value=session_mock),
patch.object(logger, "info"),
patch.object(logger, "error"),
patch.object(logger, "warn"),
):
try:
init(engine_mock)
connection_successful = True
except Exception:
connection_successful = False
assert (
connection_successful
), "The database connection should be successful and not raise an
exception."
assert session_mock.exec.called_once_with(
select(1)
), "The session should execute a select statement once."
```

init

```
def test_init_successful_connection() -> None:
engine_mock = MagicMock()
session_mock = MagicMock()
exec_mock = MagicMock(return_value=True)
session_mock.configure_mock(**{"exec.return_value": exec_mock})
with (
patch("sqlmodel.Session", return_value=session_mock),
patch.object(logger, "info"),
patch.object(logger, "error"),
patch.object(logger, "warn"),
):
try:
init(engine_mock)
connection_successful = True
except Exception:
connection_successful = False
assert (
connection_successful
), "The database connection should be successful and not raise an
exception."
assert session_mock.exec.called_once_with(
select(1)
), "The session should execute a select statement once."
```

select

```
def test_init_successful_connection() -> None:
engine_mock = MagicMock()
session_mock = MagicMock()
exec_mock = MagicMock(return_value=True)
session_mock.configure_mock(**{"exec.return_value": exec_mock})
with (
patch("sqlmodel.Session", return_value=session_mock),
patch.object(logger, "info"),
patch.object(logger, "error"),
patch.object(logger, "warn"),
):
try:
init(engine_mock)
connection_successful = True
except Exception:
connection_successful = False
assert (
connection_successful
), "The database connection should be successful and not raise an
exception."
assert session_mock.exec.called_once_with(
select(1)
), "The session should execute a select statement once."
```

hasattr

```
def test_create_user(db: Session) -> None:
email = random_email()
password = random_lower_string()
user_in = UserCreate(email=email, password=password)
user = crud.create_user(session=db, user_create=user_in)
assert user.email == email
assert hasattr(user, "hashed_password")
```

crud.authenticate

```
def test_not_authenticate_user(db: Session) -> None:
email = random_email()
password = random_lower_string()
user = crud.authenticate(session=db, email=email, password=password)
assert user is None
```

db.get

```
def test_update_user(db: Session) -> None:
password = random_lower_string()
email = random_email()
user_in = UserCreate(email=email, password=password, is_superuser=True)
user = crud.create_user(session=db, user_create=user_in)
new_password = random_lower_string()
user_in_update = UserUpdate(password=new_password, is_superuser=True)
```

```
if user.id is not None:
crud.update_user(session=db, db_user=user, user_in=user_in_update)
user_2 = db.get(User, user.id)
assert user_2
assert user.email == user_2.email
assert verify_password(new_password, user_2.hashed_password)
```

jsonable_encoder

```
def test_get_user(db: Session) -> None:
password = random_lower_string()
username = random_email()
user_in = UserCreate(email=username, password=password,
is_superuser=True)
user = crud.create_user(session=db, user_create=user_in)
user_2 = db.get(User, user.id)
assert user_2
assert user.email == user_2.email
assert jsonable_encoder(user) == jsonable_encoder(user_2)
```

verify_password

```
def test_update_user(db: Session) -> None:
password = random_lower_string()
email = random_email()
user_in = UserCreate(email=email, password=password, is_superuser=True)
user = crud.create_user(session=db, user_create=user_in)
new_password = random_lower_string()
user_in_update = UserUpdate(password=new_password, is_superuser=True)
if user.id is not None:
crud.update_user(session=db, db_user=user, user_in=user_in_update)
user_2 = db.get(User, user.id)
assert user_2
assert user.email == user_2.email
assert verify_password(new_password, user_2.hashed_password)
```

generate_password_reset_token

```
def test_reset_password(
client: TestClient, superuser_token_headers: dict[str, str]
) -> None:
token = generate_password_reset_token(email=settings.FIRST_SUPERUSER)
data = {"new_password": "changethis", "token": token}
r = client.post(
f"{settings.API_V1_STR}/reset-password/",
headers=superuser_token_headers,
json=data,
)
assert r.status_code == 200
assert r.json() == {"message": "Password updated successfully"}
```

response.json

```
def test_delete_item_not_enough_permissions(
client: TestClient, normal_user_token_headers: dict[str, str], db:
Session
) -> None:
item = create_random_item(db)
response = client.delete(
f"{settings.API_V1_STR}/items/{item.id}",
headers=normal_user_token_headers,
)
assert response.status_code == 400
content = response.json()
assert content["detail"] == "Not enough permissions"
```

create_random_item

```
def test_delete_item_not_enough_permissions(
client: TestClient, normal_user_token_headers: dict[str, str], db:
Session
) -> None:
item = create_random_item(db)
response = client.delete(
f"{settings.API_V1_STR}/items/{item.id}",
headers=normal_user_token_headers,
)
assert response.status_code == 400
content = response.json()
assert content["detail"] == "Not enough permissions"
```

client.get

```
def test_read_items(
client: TestClient, superuser_token_headers: dict[str, str], db: Session
) -> None:
create_random_item(db)
create_random_item(db)
response = client.get(
f"{settings.API_V1_STR}/items/",
headers=superuser_token_headers,
)
assert response.status_code == 200
content = response.json()
assert len(content["data"]) >= 2
```

len

```
def test_read_items(
client: TestClient, superuser_token_headers: dict[str, str], db: Session
) -> None:
create_random_item(db)
create_random_item(db)
response = client.get(
f"{settings.API_V1_STR}/items/",
headers=superuser_token_headers,
```

```
)
assert response.status_code == 200
content = response.json()
assert len(content["data"]) >= 2
```

client.put

```
def test_update_item_not_enough_permissions(
client: TestClient, normal_user_token_headers: dict[str, str], db:
Session
) -> None:
item = create_random_item(db)
data = {"title": "Updated title", "description": "Updated description"}
response = client.put(
f"{settings.API_V1_STR}/items/{item.id}",
headers=normal_user_token_headers,
json=data,
)
assert response.status_code == 400
content = response.json()
assert content["detail"] == "Not enough permissions"
```

client.delete

```
def test_delete_item_not_enough_permissions(
client: TestClient, normal_user_token_headers: dict[str, str], db:
Session
) -> None:
item = create_random_item(db)
response = client.delete(
f"{settings.API_V1_STR}/items/{item.id}",
headers=normal_user_token_headers,
)
assert response.status_code == 400
content = response.json()
assert content["detail"] == "Not enough permissions"
```

client.patch

```
def test_update_user_email_exists(
client: TestClient, superuser_token_headers: dict[str, str], db: Session
) -> None:
username = random_email()
password = random_lower_string()
user_in = UserCreate(email=username, password=password)
user = crud.create_user(session=db, user_create=user_in)
username2 = random_email()
password2 = random_lower_string()
user_in2 = UserCreate(email=username2, password=password2)
user2 = crud.create_user(session=db, user_create=user_in2)
data = {"email": user2.email}
r = client.patch(
f"{settings.API_V1_STR}/users/{user.id}",
```

```
headers=superuser_token_headers,
json=data,
)
assert r.status_code == 409
assert r.json()["detail"] == "User with this email already exists"
```

Functions

get_user_by_email

This Python function `get_user_by_email` takes in a SQLAlchemy session and an email as arguments, and using the session and the ORM (Object-Relational Mapping) tool called SQLAlchemy, it retrieves a single user from the database by matching their email with the given email. The function returns either the found user or `None` if no user is found. The returned user object is assigned to a variable named `session_user` and then passed back as the function's return value.

````
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_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 (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, or None if no such user exists.
"""
statement = select(User).where(User.email == email)
session_user = session.exec(statement).first()
return session_user
```
This commented version of the function provides more information about
its purpose, parameters, and return value than the original code. It also
includes a docstring that explains the function's behavior in detail.
````

create_user

The `create_user` function in Python creates a new user when called. It takes two arguments: `session`, which is an instance of the SQLAlchemy Session class used for database transactions, and `user_in`, which is an object containing information about the new user being created. First, it checks if a user already exists in the system with the same email address using another function called `crud.get_user_by_email`. If such a user is found, the function raises an HTTPException with a 400 status code and a detailed error message. If the new user's email address is valid and the `emails_enabled` setting is true, the function generates a new account email using another function called `generate_new_account_email`. This email contains information about the username, password, and email address of the new user. The email is then sent to the user's email address using a function called `send_email`. After all the necessary checks and preparations are completed, the new user is created using another function called `crud.create_user`. The newly created user is then returned from the `create_user` function.

Here is the commented version of the function:
```
def create_user(*, session: SessionDep, user_in: UserCreate) -> Any:
"""
Create new user.

Parameters
----------
session : SessionDep
The current session object.
user_in : UserCreate
The input user data.

Returns
-------
Any
The created user object.
"""
""" Get the user by email"""
user = crud.get_user_by_email(session=session, email=user_in.email)

""" Check if the user already exists"""
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 an email to the user's email address if enabled"""
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
```

generate_new_account_email

This Python function takes in three arguments: `email_to`, `username`, and `password`. It generates a new account email with the given user information and returns an object called `EmailData` containing the email content and subject. The project name is retrieved from a setting, and a custom HTML email

template is rendered using Django's `render_email_template` function. The resulting email will be sent to the specified email address (`email_to`) with the given username, password, and project name as context for the email template.

````
The system will generate a commented version of the function based on its
analysis. Here is an example of what the commented code might look like:
```
""" This function generates a new account email for a user with the given
username and password."""
def generate_new_account_email(
""" The email address to send the email to."""
email_to: str,

""" The username of the new account."""
username: str,

""" The password for the new account."""
password: str
) -> EmailData:
""" Load project settings from environment variables."""
settings = Settings()

""" Set the subject line of the email."""
subject = f"{settings.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 EmailData object with the rendered content and subject
line."""
return EmailData(html_content=html_content, subject=subject)
```
````

send_email

The `send_email` function in Python is used to send an email with a customizable subject, HTML content, and recipient email address. The function first checks if the necessary configuration for email variables is provided and raises an assertion error if not. Then, it creates a new email message using the `emails.Message()` class from the standard library's `email.mime.text` module. The sender name and email address are specified in the configuration settings.

The function also accepts optional arguments for the SMTP server host, port, TLS or SSL encryption

method, and authentication credentials. The message is then sent using the `send()` method of the `Message()` class with the specified recipient email address and SMTP connection options. The function returns `None` to indicate successful completion.

The function also logs the result of sending the email for future reference or debugging purposes.

````
The system parses the function to identify its purpose, parameters, and
return value. It then generates a commented version of the function based
on this analysis. The following is the commented code for the given
Python function:
```
""" Send an email with the specified subject, HTML content, and
recipient."""
def send_email(
*, # This is called a "keyword-only argument" in Python. It means that
only positional arguments can be passed after this one.
email_to: str, # Recipient of the email (must be a string)
subject: str = "", # Subject of the email (defaults to an empty string)
html_content: str = "", # HTML content of the email (defaults to an empty
string)
):
assert settings.emails_enabled, "no provided configuration for email
variables" # Ensure that emails are enabled before sending them.
message = emails.Message( # Create a new email message with the specified
subject and HTML content.
subject=subject,
html=html_content,
mail_from=(settings.EMAILS_FROM_NAME, settings.EMAILS_FROM_EMAIL),
)
smtp_options = {"host": settings.SMTP_HOST, "port": settings.SMTP_PORT}
# Set the SMTP host and port based on the settings.
if settings.SMTP_TLS: # If TLS is enabled, set the "tls" option to True.
smtp_options["tls"] = True
elif settings.SMTP_SSL: # Otherwise, if SSL is enabled, set the "ssl"
option to True.
smtp_options["ssl"] = True
if settings.SMTP_USER: # If a user has been provided, add it to the SMTP
options.
smtp_options["user"] = settings.SMTP_USER
if settings.SMTP_PASSWORD: # Similarly, if a password has been provided,
add it to the SMTP options.
smtp_options["password"] = settings.SMTP_PASSWORD
response = message.send(to=email_to, smtp=smtp_options) # Send the email
with the specified recipient and SMTP options.
logging.info(f"send email result: {response}") # Log the send email
result.
```
````

get_password_hash

This Python function, named `get_password_hash`, takes a string argument `password` and returns the hashed value of that password using the `pwd_context` module in Python's standard library. The function is called `get_password_hash` because it retrieves a hash value for a given password, which can be used for secure storage and comparison. The return type is a string, indicating that the hashed value will be provided back to the user.

```
[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("abc123") == "hashed_abc123"
""" Test case 2:"""
assert get_password_hash("password1") == "hashed_password1"
""" Test case 3:"""
assert get_password_hash("12345678") == "hashed_12345678"
[/TESTS]
```

get_user_by_email

The `get_user_by_email` function is a Python method that retrieves a user with the specified email address from a database using the SQLAlchemy ORM (Object-Relational Mapping) library. It takes two arguments: `session`, which is a session object used to manage connections to the database, and `email`, which is the email address of the user being searched for. The function returns the user found with the matching email address or `None` if no user is found. This method is marked as a coroutine using the `async def` syntax, but since we're providing an actual `Session` argument and not an async context manager, it's actually invoked synchronously.

````
Here is the commented version of the 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 SQLAlchemy session object used to query the
database.
email (str): The email address of the user to retrieve.

Returns:
````
````

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

create_user

The Python function `create_user` creates a new user. It first checks if a user already exists with the same email using the `crud.get_user_by_email` function. If such a user exists, an error is raised. Otherwise, a new user is created using the `crud.create_user` function and passed the user input (stored in the `user_in` argument). Additionally, if email notifications are enabled and an email address was provided during user creation, an email is generated using the `generate_new_account_email` function and sent using the `send_email` function. The finished user object is then returned from the function.

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

user = crud.create_user(session=session, user_create=user_in)
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
```
```
The output of the system will be:
```
```
""" create_user(*, session: SessionDep, user_in: UserCreate) -> Any:"""
""" """""""
""" Create new user."""
""" """""""
```

```
""" 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.","""
""" )"""
""" """
""" user = crud.create_user(session=session, user_create=user_in)"""
""" 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"""
```

get_password_hash

This Python function, named `get_password_hash`, takes a password as an argument of type string and returns its hash using the `pwd_context` module. The `pwd_context` is likely referring to the `Crypt` library in Python, which provides secure password storage methods. This function can be used for securely storing passwords in databases or files instead of plain text versions.

The system will parse 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:

```
""" Function to get password hash"""
def get_password_hash(password: str) -> str:
"""
Generates a password hash using the pwd_context module.

:param password: The input password as a string.
:return: A generated password hash as a string.
"""
return pwd_context.hash(password)
```

In this example, the system has identified that the function takes a single parameter `password` of type `str`, and returns a value of type `str`. The system has also added a docstring to explain the purpose of the function and provide information about its parameters and return value.

generate_test_email

The function `generate_test_email` takes a string `email_to` as input and returns an instance of `EmailData`. The function sets the project name from the `settings.PROJECT_NAME` variable, creates a subject for the email with this project name and "Test email" appended, and generates HTML content using the `render_email_template` function with the template named `test_email.html` and a context containing the project name and email address to be replaced in the template. The `EmailData` object is then returned with both the generated HTML content and subject. In summary, this function generates an email with a test message containing the project name and sends it to the specified email address using the provided settings.

````
Here is a commented version of the `generate_test_email` function:
```
def generate_test_email(email_to: str) -> EmailData:
"""
Generate a test email using the provided recipient email address.

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

Returns:
EmailData: An object containing the HTML content and subject of the test
email.
"""
project_name = settings.PROJECT_NAME # Get the current project name from
settings
subject = f"{project_name} - Test email" # Create a subject line for the
test email
html_content = render_email_template( # Render an HTML template with the
project name and recipient email address
template_name="test_email.html",
context={"project_name": settings.PROJECT_NAME, "email": email_to},
)
return EmailData(html_content=html_content, subject=subject) # Return an
object containing the HTML content and subject of the test email
```
In this comment, we have added a brief description of what the function
does, as well as any parameters or return values that are relevant to the
function's purpose. We have also used natural language to explain the
code, making it easier for other developers to understand the function's
intent and usage.
````

send_email

This Python function, named `send_email`, sends an email with a customizable subject, HTML content, and recipient address (specified by the `email_to` parameter). The configuration for email variables is required (as indicated by the `settings.emails_enabled` assertion), and the email's sender name and email address can be set through the `settings` object. SMTP settings such as TLS, SSL, username, and password can also be specified using the `settings` object. The function returns `None`, and its execution is logged to the console for debugging purposes.

Here's the commented version of the function based on the analysis:
```
def send_email(
*, # This parameter is used to pass any number of positional arguments
email_to: str, # The recipient of the email
subject: str = "", # The subject line of the email (defaults to an empty string)
html_content: str = "", # The HTML content of the email (defaults to an empty string)
) -> None:
"""Send an email using the configured SMTP settings.

Args:
email_to (str): The recipient of the email
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 is returned

Raises:
AssertionError: If the settings for email variables are not provided

Examples:
>>> send_email(email_to="john.doe@example.com", subject="Test Subject", html_content="Hello, world!")
send email result: {'ok': True}
```
The comments explain the purpose of the function, the parameters and their default values, the return value, and any potential exceptions that may be raised. The examples section provides a usage example for the function.

authenticate

The `authenticate` function in Python takes arguments `session`, `email`, and `password`. It first retrieves the user with the provided email from a database using the `get_user_by_email` function. If the user is not found, the function returns `None`. Otherwise, it checks whether the provided password matches the hashed password of the user using the `verify_password` function. If the passwords do not match, the function returns `None`. If both conditions are met, the function returns the authenticated user object. Overall, this function is used to authenticate a user's login credentials against a database.

Here is the commented version of the function based on the analysis:
```
def authenticate(*, session: Session, email: str, password: str) -> User | None:
"""
Authenticates a user with their email and password. If successful, returns the User object, otherwise returns None.
```

```
Parameters:
* session (Session): The database session used for querying the user
data.
* email (str): The user's email address.
* password (str): The user's password.

Returns:
User | None: The authenticated user object, 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

The function `create_access_token` in Python takes two arguments - `subject`, which can be a string or any other type, and `expires_delta`, which is a time delta. It creates an access token by encoding a dictionary containing the `exp` (expiration time) and `sub` (subject) fields using the `jwt.encode()` function with the provided secret key and algorithm. The expiration time is calculated as the current UTC time plus the specified time delta, and the subject is converted to a string before being included in the dictionary. The encoded access token is then returned by the function. In summary, this function generates an access token with an expiration time and includes specific information about the request's subject.

```
Here is a commented version of the `create_access_token` function based
on the analysis:
```
```
def create_access_token(subject: str | Any, expires_delta: timedelta) ->
str:
"""
Generates an access token for the given subject and expiration delta.

Parameters:
subject (str | Any): The subject of the access token. Can be a string or
any object.
expires_delta (timedelta): The amount of time the access token is valid
for.

Returns:
str: The encoded JWT access token.
"""
""" Get the current UTC datetime"""
now = datetime.utcnow()

""" Calculate the expiration date and time"""
expire = now + expires_delta
```

```
""" Create a dictionary with the "exp" and "sub" claims"""
to_encode = {
"exp": expire,
"sub": str(subject)
}

""" Encode the JWT using the SECRET_KEY and ALGORITHM settings"""
encoded_jwt = jwt.encode(to_encode, settings.SECRET_KEY,
algorithm=ALGORITHM)

""" Return the encoded JWT access token"""
return encoded_jwt
```

get_user_by_email

The function `get_user_by_email` is a method that retrieves a user with the provided email address using the SQLAlchemy library's session object. It takes two parameters, `session` and `email`, where `session` is an instance of the SQLAlchemy Session class, and `email` is the unique identifier for the user in question. The function uses a SELECT statement with a WHERE clause to filter the User table for the user with the specified email address. The resultant row from the query is returned as either a User object or None if no matching user is found.

```
The system will parse the function to identify its purpose, parameters,
and return value. Based on this analysis, it will generate a commented
version of the function as follows:
```
```
def get_user_by_email(*, session: Session, email: str) -> User | None:
"""
Retrieves a user by their email address.

:param session: The SQLAlchemy session to use for the query.
:type session: sqlalchemy.orm.Session
:param email: The email address of the user to retrieve.
:type email: str
:return: The retrieved user, or None if no user was found with the given
email.
:rtype: User | None
"""
statement = select(User).where(User.email == email)
session_user = session.exec(statement).first()
return session_user
```
```
This commented version of the function provides a description of its
purpose, lists each of its parameters and their types, and specifies the
expected return value. It also includes a colon (`:`) after each
parameter name to indicate that it is a type hint for that parameter.
```

verify_password

This Python function named `verify_password` takes two arguments, a plain text password `plain_password` and a previously hashed password `hashed_password`. It then calls the `verify()` method of an instance called `pwd_context`, which is assumed to have been initialized elsewhere. This method returns a boolean value indicating whether the plain text password matches the hashed password, and the function returns this value as well. In summary, this function checks if a given plain text password matches the hashed version of that password.

```
```
""" Verify password"""
"""""""
""" This function takes in two parameters:"""
""" - plain_password: the plain text password to be verified"""
""" - hashed_password: the previously hashed password to compare with"""
"""""""
""" It returns a boolean value indicating whether the provided plain
password matches the previously hashed password"""
def verify_password(plain_password: str, hashed_password: str) -> bool:
""" Verify the password by comparing it with the previously hashed
password"""
return pwd_context.verify(plain_password, hashed_password)
```
```

get_user_by_email

This Python function `get_user_by_email` is a method that retrieves a user by their email address from a database using the SQLAlchemy library. The function takes two arguments, `session` which represents the current database session, and `email`, the unique identifier for the user's email address.

The function uses the built-in SQLAlchemy method `select` to create a statement that selects all columns from the `User` table where the value of the `email` column matches the provided `email` argument. The statement is executed using the `exec` method, and the resulting data (if any) is returned as the `session_user` variable.

Finally, the function returns either the retrieved user object or `None` if no user was found in the database for the specified email address.

```
```
""" Function to get a user by their email address"""
def get_user_by_email(*, session: Session, email: str) -> User | None:
""" SQL statement to select the user based on their email address"""
statement = select(User).where(User.email == email)

""" Execute the SQL statement and fetch the first result"""
session_user = session.exec(statement).first()

""" Return the user object or None if no user was found"""
return session_user
```
```

generate_password_reset_token

This Python function generates a password reset token for the provided email address. It creates an expiration time based on the number of hours specified in a settings variable, calculates the current UTC time, adds the delta to get the expiration time, and then uses the JWT (JSON Web Tokens) library to encode a payload containing the expiration time, creation time, and email address into a string token. The function returns the generated token.

````
The system will generate a commented version of the function based on its
analysis:
```
def generate_password_reset_token(email: str) -> str:
"""
Generates a password reset token for the specified email address.

Parameters:
email (str): The email address to generate the reset token for.

Returns:
encoded_jwt (str): The generated reset token, which is an encrypted JSON
web token.
"""
delta = timedelta(hours=settings.EMAIL_RESET_TOKEN_EXPIRE_HOURS)
now = datetime.utcnow()
expires = now + delta
exp = expires.timestamp()
encoded_jwt = jwt.encode(
{"exp": exp, "nbf": now, "sub": email},
settings.SECRET_KEY,
algorithm="HS256",
)
return encoded_jwt
```
````

generate_reset_password_email

The Python function `generate_reset_password_email` takes three arguments: `email_to`, `email`, and `token`. It then retrieves the project name from a setting called `PROJECT_NAME`, generates an email subject with the user's email and project name, creates a link with the token for resetting password, renders an email template named "reset_password.html" using Django's `render_email_template` function, and returns an object of type `EmailData` containing the HTML content of the email and subject. This function is likely used to send an email to a user requesting a password reset with a link to click that will take them to a page where they can enter a new password. The function's purpose is to generate the email content automatically based on the provided arguments, making it easier to reuse this functionality in other parts of the application.

````
The system will generate a commented version of the function based on its
analysis. The generated code will be similar to the following:
```
def generate_reset_password_email(email_to: str, email: str, token: str)
-> EmailData: # Purpose of this function is to generate an email for
````
````

```
password recovery.
project_name = settings.PROJECT_NAME # Name of the project from where the
email is being sent.
subject = f"{project_name} - Password recovery for user {email}" #
Subject of the email, includes the name of the project and the username
of the person who requested the password reset.
link = f"{settings.server_host}/reset-password?token={token}" # URL to
which the email link will redirect to.
html_content = render_email_template( # Function that renders the HTML
template for the email content.
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) # Return
value of the function, which includes the HTML content and the subject of
the email.
```
In this example, the system has identified the purpose of the function as generating an email for password recovery, and has generated a commented version of the code based on this analysis. The comments explain each line of code in detail, making it easier to understand the function's intended behavior.

send_email

The `send_email` function in Python is used to send an email with a customizable subject and optional HTML content to a specified recipient, defined by the `email_to` parameter. If no subject or HTML content are provided, they will be empty strings by default. The function first checks if configuration settings for email variables are provided and raises an assertion error if not. It then creates a new message object using the `emails.Message()` class from an external library called "emails". The SMTP (Simple Mail Transfer Protocol) options, such as host, port, TLS/SSL encryption, user, and password, are defined in configuration settings. After creating the message object, it is sent to the recipient using the `send()` method of the message object, with the necessary options passed as arguments. The result of the email sending process is logged for information purposes. Overall, this function provides a customizable way to send emails from within Python code with different levels of configuration settings provided.

```
Sure! Here is the commented version of the `send_email` function based on
the analysis:
```
```
def send_email(
""" Sends an email to the specified recipient with the given subject and
content."""
*,
email_to: str,
subject: str = "",
```

```
html_content: str = "",
) -> None:
""" Assert that email settings are enabled, otherwise throw an error."""
assert settings.emails_enabled, "no provided configuration for email
variables"

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

""" Define the SMTP options for sending the email."""
smtp_options = {"host": settings.SMTP_HOST, "port": settings.SMTP_PORT}

""" Check if TLS or SSL is enabled and add it to the SMTP options if
necessary."""
if settings.SMTP_TLS:
smtp_options["tls"] = True
elif settings.SMTP_SSL:
smtp_options["ssl"] = True

""" Check if a username or password is specified and add it to the SMTP
options if necessary."""
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 specified SMTP options."""
response = message.send(to=email_to, smtp=smtp_options)

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

verify_password_reset_token

This Python function `verify_password_reset_token` takes a string `token` as input and returns either a string or `None`. It first attempts to decode the `token` using the `jwt.decode()` function with the specified secret key and algorithm ("HS256"). If the decoding is successful, it extracts the value of the "sub" (subject) field from the decoded token and returns it as a string. If the decoding fails due to an error (such as an incorrect token or expiration), `None` is returned instead.

The system will generate the following commented version of the function:

```
def verify_password_reset_token(token: str) -> str | None:
"""
Verifies a password reset token and returns the user ID if valid, or None
```

```
otherwise.

Parameters:
token (str): The password reset token to be verified.

Returns:
str | None: The user ID if the token is valid, or None if it's invalid or
has expired.
"""
try:
decoded_token = jwt.decode(token, settings.SECRET_KEY,
algorithms=["HS256"])
return str(decoded_token["sub"])
except JWTError:
return None
```

get_user_by_email

This Python function is called `get_user_by_email` and takes two arguments, a SQLAlchemy session (`session`) and the user's email address (`email`). It returns either the User object from the database corresponding to the given email or `None` if no such User exists. The function creates a SELECT statement using SQLAlchemy's `select()` function, filters the result set to only include Users with the specified email address using the `where()` function, executes the query within the session object using `exec()`, and returns the first result returned by the query as the session_user variable.

```
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_user_by_email(*, session: Session, email: str) -> User | None:
"""
Returns a user object by email address.

Args:
* session: A SQLAlchemy session object used to query the database.
* email: The email address of the user to retrieve.

Returns:
A User object if found, or None if not found.
"""

statement = select(User).where(User.email == email)
session_user = session.exec(statement).first()
return session_user
```

get_password_hash

The given Python function, named `get_password_hash`, takes a single argument `password` of type string and returns a string representing the hash value of the password using the `pwd_context` module for password storage. This hashed value can be used to securely store the password in a database or other system without storing the plain text password itself, improving security.

```
```
""" Get password hash"""
"""""""
""" Purpose: Generates a hash of the given password using the pwd_context
context."""
"""""""
""" Parameters:"""
""" password (str): The password to be hashed."""
"""""""
""" Return value:"""
""" str: The generated hash of the password."""
def get_password_hash(password: str) -> str:
return pwd_context.hash(password)
```
```

get_user_by_email

This Python function, named `get_user_by_email`, takes two arguments: a database session (`session`) and an email address (`email`). It retrieves a user record from the database by selecting (`select`) the `User` object where the `email` matches the input value (`where(User.email == email)`). The result is returned as either the selected `User` object or `None` if no match was found (`session.exec(statement).first()`). This function can be used to efficiently and securely retrieve a user's details based on their email address, which may be useful for various operations such as login, password reset, or account management.

```
The system will generate a commented version of the function as follows:
```
def get_user_by_email(*, session: Session, email: str) -> User | None:
""" Select a user from the database based on their email address"""
statement = select(User).where(User.email == email)

""" Execute the query and retrieve the first result"""
session_user = session.exec(statement).first()

return session_user
```
```

generate_password_reset_token

The provided Python function `generate_password_reset_token` takes an email as input and returns a string representing a JWT (JSON Web Token) used for password reset operations. The function creates a token that expires after a specified number of hours (determined by the `settings.EMAIL_RESET_TOKEN_EXPIRE_HOURS` variable), with a payload containing the email address and an expiration timestamp generated from the current UTC time and the specified time delta. The JWT is encoded using the `jwt.encode()` function, which requires the secret key (stored in the

`settings.SECRET_KEY`) and the HS256 encryption algorithm.

````
Here's the commented version of the function:
```
def generate_password_reset_token(email: str) -> str:
"""
Generates a password reset token for the specified email address.

:param email: The email address to generate the token for.
:return: A signed, JSON Web Token (JWT) that contains the expiration date
and the email address.
"""
""" Calculate the delta between now and the expiration date"""
delta = timedelta(hours=settings.EMAIL_RESET_TOKEN_EXPIRE_HOURS)
now = datetime.utcnow()
expires = now + delta
exp = expires.timestamp()
""" Encode the JWT with the specified algorithm"""
encoded_jwt = jwt.encode(
{"exp": exp, "nbf": now, "sub": email},
settings.SECRET_KEY,
algorithm="HS256",
)
return encoded_jwt
```
````

generate_reset_password_email

This Python function named `generate_reset_password_email` takes three arguments: `email_to`, `email`, and `token`. It then uses the `settings` object to retrieve the project name and server host, and creates a subject for the email with this information. The link to reset the password is generated using the `token` parameter and the server host.

Next, the function renders an email template called "reset_password.html" using the `render_email_template` helper method. It passes the project name, username (email), email address to send the email to, the number of valid hours for the token, and the generated link as context to the template.

Finally, the function returns an object called `EmailData` containing the HTML content and subject of the email that was just created. This object can be used by an email sending library or framework to actually send the password reset email. Overall, this function generates a password reset email with a customizable subject, link, and template for sending out via email.

````
The system generates the following commented version of the function:
```
def generate_reset_password_email(email_to: str, email: str, token: str)
-> EmailData:
""" Purpose: Generate an email that allows a user to reset their
password."""
```
````

```
""" Parameters:"""
""" - email_to (str): The email address of the user who is requesting the
reset."""
""" - email (str): The email address of the user whose password needs to
be reset."""
""" - token (str): A unique token used for password recovery."""

""" Return value: An instance of EmailData that contains 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)
```