Bunker Management System

Release 1.0

NEXUS 2

Dec 15, 2024

CONTENTS:

CHAPTER

ONE

CONTROLLERS PACKAGE

1.1 admin_controller module

class app.controllers.admin_controller.**AdminController**(*db_url=None*)

Bases: object

create_admin(*admin_data:* Admin, *session=None*)

Creates a new admin in the system.

This method checks if an admin with the provided email already exists. If not, it creates a new admin record in the database with the provided details.

Parameters

- **admin_data** (*AdminModel*) – An object containing the admin's details, including: - email (str): The admin's email address. - name (str): The admin's name. - password (str): The admin's hashed password.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "message": "Admin created successfully."} if the admin is created successfully.
- {"status": "error", "message": <error_message>} if there is an error, such as an existing email or a database issue.

Return type
dict

Raises
Exception – If any unexpected error occurs during the creation process.

deleteAdmin(*admin_id: int*, *session=None*)

Deletes an admin from the database using their ID.

This method looks up an admin in the database by their unique ID and deletes the corresponding record if it exists.

Parameters

- **admin_id** (*int*) – The unique identifier of the admin to delete.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “message”: “Admin deleted successfully”}:

If the admin is successfully deleted. - {“status”: “error”, “message”: <error_message>}: If an error occurs or the admin is not found.

Return type
dict

Raises
Exception – If an unexpected error occurs during the deletion process.

getAdminById(*idAdmin: int*, *session=None*)

Retrieves an admin by their ID.

This method queries the database for an admin with the specified ID and returns their details if found.

Parameters

- **idAdmin** (*int*) – The unique identifier of the admin to retrieve.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “admin”: <admin_info>}:

If the admin is found. *admin_info* contains:

- idAdmin (int): The unique identifier of the admin.
- email (str): The admin’s email address.
- name (str): The admin’s name.
- password (str): The admin’s hashed password.

- {“status”: “error”, “message”: <error_message>}:

If an error occurs or the admin is not found.

Return type
dict

Raises
Exception – If an unexpected error occurs while querying the database.

listAdmins(*session=None*)

Lists all admins registered in the database.

This method retrieves all admin records from the database and formats them into a list of dictionaries containing their ID and email.

Parameters
session (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “admins”: [<list_of_admins>]}:

If the admins are successfully retrieved. Each admin in the list contains:

- idAdmin (int): The unique identifier of the admin.
- email (str): The email address of the admin.

- {“status”: “error”, “message”: <error_message>}:

If an error occurs during the operation.

Return type
dict

Raises
Exception – If an unexpected error occurs while querying the database.

loginAdmin(*email: str*, *password: str*, *session=None*)

Verifies the login credentials of an admin using their email and password.

Parameters

- **email** (*str*) – The admin’s email address.
- **password** (*str*) – The admin’s plain text password.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction.

Returns

Result of the login attempt.

- {“status”: “ok”, “token”: <JWT token>, “user”: {“idAdmin”: <idAdmin>, “email”: <email>}}:

If the login is successful. - {“status”: “error”, “message”: <error_message>}: If an error occurs or the credentials are invalid.

Return type
dict

refreshAccessToken(*refresh_token: str*)

updateAdminEmail(*idAdmin: int*, *new_email: str*, *session=None*)

Updates the email of an admin.

This method searches for an admin by their ID and updates their email to the provided new value.

Parameters

- **idAdmin** (*int*) – The unique identifier of the admin whose email will be updated.
- **new_email** (*str*) – The new email address to assign to the admin.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “message”: “Email updated successfully”}:

If the email is updated successfully. - {“status”: “error”, “message”: <error_message>}: If an error occurs or the admin is not found.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

updateAdminName(*idAdmin: int*, *new_name: str*, *session=None*)

Updates the name of an admin.

This method searches for an admin by their ID and updates their name to the provided new value.

Parameters

- **idAdmin** (*int*) – The unique identifier of the admin whose name will be updated.
- **new_name** (*str*) – The new name to assign to the admin.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “message”: “Name updated successfully”}:

If the name is updated successfully. - {“status”: “error”, “message”: <error_message>}: If an error occurs or the admin is not found.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

updateAdminPassword(*idAdmin: int*, *new_password: str*, *session=None*)

Updates the password of an admin.

This method searches for an admin by their ID and updates their password to the provided new value.

Parameters

- **idAdmin** (*int*) – The unique identifier of the admin whose password will be updated.
- **new_password** (*str*) – The new password to assign to the admin.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “message”: “Password updated successfully”}:

If the password is updated successfully. - {“status”: “error”, “message”: <error_message>}: If an error occurs or the admin is not found.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

1.2 alarm_controller module

class app.controllers.alarm_controller.**AlarmController**(*db_url=None*)

Bases: object

create_alarm(*body:* Alarm, *session=None*)

Creates a new alarm in the database.

This method validates the existence of a room and ensures that no duplicate alarms exist in the same room before creating a new alarm.

Parameters

- **body** (*AlarmModel*) – An object containing the details of the alarm to be created, including: - idAlarm (int): The unique identifier of the alarm. - start (datetime): The start time of the alarm. - end (datetime): The end time of the alarm. - idRoom (int): The unique identifier of the room where the alarm is created. - createDate (datetime): The date when the alarm is created.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”}:

If the alarm is created successfully. - {“status”: “error”, “message”: <error_message>}: If an error occurs, such as the room not existing or a duplicate alarm.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

create_alarmLevel(*body:* Alarm, *session=None*)

Creates a new alarm with an automatically generated *idAlarm*.

This method assigns the alarm to room 3 and validates its existence before creating the alarm. The *idAlarm* is generated automatically by the database if the column is set as auto-increment.

Parameters

- **body** (*AlarmModel*) – An object containing the details of the alarm to be created, including: - start (datetime): The start time of the alarm. - end (datetime, optional): The end time of the alarm (can be None initially). - createDate (datetime): The date when the alarm is created.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “message”: “Alarm <idAlarm> created successfully in room 3.”, “idAlarm”: <idAlarm>}:

If the alarm is created successfully. Includes the generated *idAlarm*. - {“status”: “error”, “message”: <error_message>}: If an error occurs, such as room 3 not existing or a database issue.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

list_alarms(*session=None*)

Lists all alarms in the database.

This method retrieves all alarms from the database and returns their details as a list of dictionaries.

Parameters
session (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “alarms”: [<list_of_alarms>]}:

If the alarms are successfully retrieved. Each alarm in the list contains:

- idAlarm (int): The unique identifier of the alarm.
- start (str, optional): The start time of the alarm in ISO 8601 format.
- end (str, optional): The end time of the alarm in ISO 8601 format.
- idRoom (int): The unique identifier of the room associated with the alarm.
- createDate (str, optional): The creation date of the alarm in ISO 8601 format.

- {“status”: “ok”, “alarms”: []}:

If no alarms are found in the database. - {“status”: “error”, “message”: <error_message>}: If an error occurs during the operation.

Return type
dict

Raises
Exception – If an unexpected error occurs while querying the database.

updateAlarmEndDate(*idAlarm: int*, *new_enddate: datetime*, *session=None*)

Updates the end date of an alarm.

This method searches for an alarm by its ID and updates its *enddate* field to the specified new value.

Parameters

- **idAlarm** (*int*) – The unique identifier of the alarm to update.
- **new_enddate** (*datetime*) – The new end date to assign to the alarm.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "message": "Alarm end date updated successfully"}:

If the end date is updated successfully. - {"status": "error", "message": <error_message>}: If an error occurs or the alarm is not found.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

1.3 family_controller module

class app.controllers.family_controller.**FamilyController**(*db_url=None*)

Bases: object

create_family(*body:* Family, *session=None*)

Creates a new family in the database and validates preconditions.

This method verifies that the room, shelter, and admin associated with the family exist, and ensures no duplicate family with the same name exists in the specified room before creating a new family.

Parameters

- **body** (*FamilyModel*) – An object containing the details of the family to be created, including: - idFamily (int): The unique identifier of the family. - familyName (str): The name of the family. - idRoom (int): The unique identifier of the room where the family resides. - idShelter (int): The unique identifier of the shelter associated with the family. - createdBy (int): The ID of the admin who creates the family record. - createDate (datetime): The date when the family record is created.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok"}:

If the family is created successfully. - {"status": "error", "message": <error_message>}: If an error occurs, such as missing entities (room, shelter, admin) or duplicate family names in the same room.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.

- **Exception** – If an unexpected error occurs during the operation.

deleteFamily(*family_id: int*, *session=None*)

Deletes a family from the database using its ID, if no members are associated with it.

This method first verifies that the family exists and that no residents are linked to the family. If there are associated members, the deletion is not allowed.

Parameters

- **family_id** (*int*) – The unique identifier of the family to delete.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “message”: “Family deleted successfully”}:

If the family is successfully deleted. - {“status”: “error”, “message”: <error_message>}: If an error occurs, such as the family not being found or members being associated with it.

Return type
dict

Raises
Exception – If an unexpected error occurs during the operation.

listFamilies(*session=None*)

Lists all families registered in the database.

This method retrieves all families from the database and formats their details into a list of dictionaries.

Parameters
session (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “families”: [<list_of_families>]}:

If the families are successfully retrieved. Each family in the list contains:

- idFamily (int): The unique identifier of the family.
- familyName (str): The name of the family.
- idRoom (int): The unique identifier of the room associated with the family.
- idShelter (int): The unique identifier of the shelter associated with the family.
- createdBy (int): The ID of the admin who created the family.
- createDate (str, optional): The date the family was created, in ISO 8601 format.

- {“status”: “error”, “message”: <error_message>}:

If an error occurs during the operation.

Return type
dict

Raises

Exception – If an unexpected error occurs while querying the database.

1.4 handler module

1.5 machine_controller module

class app.controllers.machine_controller.**MachineController**(*db_url=None*)

Bases: object

create_machine(*body:* Machine, *session=None*)

Creates a new machine entry in the database.

This method ensures the following: - The machine is assigned to an existing room. - No other machine with the same name exists in the same room. - Prevents duplication of machines within the same room.

Parameters

- **body** (*MachineModel*) – The data of the machine to be added, including: - idMachine (int): The unique identifier of the machine. - machineName (str): The name of the machine. - on (bool): The initial status of the machine (on/off). - idRoom (int): The ID of the room where the machine is located. - createdBy (int): The ID of the admin who created the machine. - createDate (datetime): The creation timestamp of the machine. - update (datetime): The last update timestamp of the machine.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”}:

If the machine is created successfully. - {“status”: “error”, “message”: <error_message>}: If an error occurs, such as the room or admin not existing, or a duplicate machine name in the same room.

Return type

dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

deleteMachine(*machine_id: int*, *session=None*)

Deletes a machine from the database using its ID.

This method searches for a machine by its unique ID and deletes it if found.

Parameters

- **machine_id** (*int*) – The unique identifier of the machine to delete.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “message”: “Machine deleted successfully”}:

If the machine is successfully deleted. - {“status”: “error”, “message”: <error_message>}: If an error occurs, such as the machine not being found.

Return type
dict

Raises
Exception – If an unexpected error occurs during the operation.

list_machines(*session=None*)

Lists all machines in the database.

This method retrieves all machines from the database and formats their details into a list of dictionaries.

Parameters
session (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “machines”: [<list_of_machines>]}:

If the machines are successfully retrieved. Each machine in the list contains:

- idMachine (int): The unique identifier of the machine.
- machineName (str): The name of the machine.
- on (bool): The current status of the machine (True for on, False for off).
- idRoom (int): The unique identifier of the room where the machine is located.
- createdBy (int): The ID of the admin who created the machine.
- createDate (str, optional): The creation date of the machine, in ISO 8601 format.
- update (str, optional): The last update timestamp of the machine, in ISO 8601 format.

- {“status”: “ok”, “machines”: []}:

If no machines are found in the database. - {“status”: “error”, “message”: <error_message>}: If an error occurs during the operation.

Return type
dict

Raises
Exception – If an unexpected error occurs while querying the database.

updateMachineDate(*machine_name: str*, *session=None*)

Updates the *update* field of a specific machine with the current date.

This method searches for a machine by its name and updates its *update* field to the current date.

Parameters

- **machine_name** (*str*) – The name of the machine to update.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “message”: “Machine date updated successfully”}:

If the machine’s date is updated successfully. - {“status”: “error”, “message”: <error_message>}: If an error occurs, such as the machine not being found.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

updateMachineStatus(*machine_name: str*, *session=None*)

Updates the status of a specific machine to *False*.

This method searches for a machine by its name and sets its *on* status to *False* to indicate that the machine is turned off.

Parameters

- **machine_name** (*str*) – The name of the machine to update.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “message”: “Machine status updated successfully”}:

If the machine’s status is updated successfully. - {“status”: “error”, “message”: <error_message>}: If an error occurs, such as the machine not being found or a database issue.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

updateMachineStatusOn(*machine_name: str*, *session=None*)

Updates the status of a machine to “on” (True).

This method searches for a machine by its name in the database, updates its *on* status to *True*, and commits the change. If the machine is not found or if an error occurs during the operation, an appropriate error message is returned.

Parameters

- **machine_name** (*str*) – The name of the machine to update.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- **{"status": "ok", "message": <success_message>}:**
  If the machine's status is successfully updated. For example: {"status": "ok", "message": "Machine 'MachineName' status updated to True"}.
- **{"status": "error", "message": "Machine '<machine_name>' not found"}:**
  If the specified machine does not exist in the database.
- **{"status": "error", "message": "Database error: <error_message>"}:**
  If an SQLAlchemy error occurs during the operation.
- **{"status": "error", "message": <error_message>}:**
  If any other unexpected error occurs during the operation.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-specific error occurs.
- **Exception** – For any other unexpected errors during execution.

Notes

- The *machine_name* must match exactly with the *machineName* field in the database.
- If the machine is not found, no changes will be made, and an error message will be returned.
- The session is closed automatically after the operation, regardless of success or failure.

1.6 resident_controller module

class app.controllers.resident_controller.**ResidentController**(*db_url=None*)

Bases: object

create_resident(*body:* Resident, *session=None*) → dict

Creates a new resident in the database.

This method verifies that the associated family, room, and shelter exist and have sufficient capacity before creating a new resident. If the room is full, a new room is created automatically and assigned to the family.

Parameters

- **body** (*ResidentModel*) – The resident data to be added, including: - name (str): The first name of the resident. - surname (str): The last name of the resident. - birthDate (date): The birthdate of the resident. - gender (str): The gender of the resident. - createdBy (int): The ID of the admin who created the resident record. - idFamily (int): The ID of the family the resident belongs to.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok"}:

If the resident is successfully created. - {"status": "error", "message": <error_message>}: If an error occurs, such as missing entities, room or shelter capacity issues, or duplicate residents.

Return type
dict

Raises
Exception – If an unexpected error occurs during the operation.

Notes

- If the family does not have an assigned room or the room is full, a new room is created

and assigned to the family. - The shelter and room capacities are validated to prevent exceeding their maximum limits.

delete_resident(*idResident: int*, *session=None*)

Deletes a resident entry from the database.

This method searches for a resident by their unique ID and deletes the record if found.

Parameters

- **idResident** (*int*) – The unique identifier of the resident to delete.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”}:

If the resident is successfully deleted. - {“status”: “not found”}: If no resident is found with the provided ID.

Return type
dict

Raises
Exception – If an unexpected error occurs during the operation.

Process:

1. Check if a database session is provided; if not, create a new session.
2. Query the database for a resident matching the provided *idResident*.
3. **If the resident exists:**
   - Delete the resident record.
   - Commit the transaction to save changes.
   - Return a success status.
4. If the resident does not exist, return a “not found” status.

getResidentById(*idResident: int*, *session=None*)

Retrieves a resident by their ID.

This method searches for a resident using their unique ID and returns their details if found.

Parameters

- **idResident** (*int*) – The unique identifier of the resident to retrieve.

- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "resident": <resident_info>}:

If the resident is found. *resident_info* contains:

- idResident (int): The unique identifier of the resident.
- name (str): The first name of the resident.
- surname (str): The last name of the resident.
- birthDate (date): The birth date of the resident.
- gender (str): The gender of the resident.
- idFamily (int): The ID of the family the resident belongs to.

- {"status": "error", "message": <error_message>}:

If an error occurs or the resident is not found.

Return type
dict

Raises
Exception – If an unexpected error occurs during the operation.

getResidentRoomByNameAndSurname(*name: str*, *surname: str*, *session=None*)

Retrieves the room where a resident is located based on their name and surname.

This method searches for a resident using their name and surname, then retrieves the details of the room they are assigned to, if available.

Parameters

- **name** (*str*) – The first name of the resident to search for.
- **surname** (*str*) – The last name of the resident to search for.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "room": <room_info>}:

If the resident and their room are successfully found. *room_info* contains:

- roomName (str): The name of the room.
- maxPeople (int): The maximum capacity of the room.
- idShelter (int): The ID of the shelter the room belongs to.
- createDate (datetime): The creation date of the room.
- createdBy (int): The ID of the admin who created the room.

- {“status”: “error”, “message”: “Resident not found”}:

If the resident is not found in the database. - {“status”: “error”, “message”: “Resident has no assigned room”}: If the resident does not have a room assigned. - {“status”: “error”, “message”: “Room not found”}: If the room assigned to the resident does not exist in the database. - {“status”: “error”, “message”: <error_message>}: If any other error occurs during the operation.

Return type
dict

Raises
Exception – If an unexpected error occurs during the operation.

list_residents(*session=None*)

Lists all residents in the database.

This method retrieves all residents from the database and returns their details in a structured format.

Parameters
session (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “residents”: [<list_of_residents>]}:

If residents are successfully retrieved. Each resident in the list contains:

- idResident (int): The unique identifier of the resident.
- name (str): The first name of the resident.
- surname (str): The last name of the resident.
- birthDate (str, optional): The birthdate of the resident in ISO 8601 format.
- gender (str): The gender of the resident.
- idFamily (int): The ID of the family the resident belongs to.
- idRoom (int): The ID of the room the resident is assigned to.

- {“status”: “ok”, “residents”: []}:

If no residents are found in the database. - {“status”: “error”, “message”: <error_message>}: If an error occurs during the operation.

Return type
dict

Raises
Exception – If an unexpected error occurs while querying the database.

list_residents_in_room(*idRoom*, *session=None*)

Lists all residents in a specific room.

This method retrieves all residents assigned to a given room and returns their details in a structured format.

Parameters

- **idRoom** (*int*) – The unique identifier of the room to list residents for.

- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "residents": [<list_of_residents>]}:

If residents are successfully retrieved. Each resident in the list contains:

- idResident (int): The unique identifier of the resident.
- name (str): The first name of the resident.
- surname (str): The last name of the resident.
- birthDate (str, optional): The birthdate of the resident in ISO 8601 format.
- gender (str): The gender of the resident.
- idFamily (int): The ID of the family the resident belongs to.
- idRoom (int): The ID of the room the resident is assigned to.

- {"status": "ok", "residents": []}:

If no residents are found in the specified room. - {"status": "error", "message": <error_message>}: If an error occurs during the operation.

Return type
dict

Raises
Exception – If an unexpected error occurs while querying the database.

login(*name: str*, *surname: str*, *session=None*)

Verifies the login credentials using the resident's name and surname.

This method checks if a resident with the provided name and surname exists in the database and returns their details if the login is successful.

Parameters

- **name** (*str*) – The first name of the resident.
- **surname** (*str*) – The last name of the resident.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "user": {"idResident": <id>, "name": <name>, "surname": <surname>}}:

If the login is successful. Contains the resident's ID, name, and surname. - {"status": "error", "message": "Invalid credentials"}: If the provided credentials do not match any resident in the database. - {"status": "error", "message": <error_message>}: If an unexpected error occurs.

Return type
dict

Raises
Exception – If an unexpected error occurs during the operation.

updateResidentBirthDate(*idResident: int*, *new_birthDate: str*, *session=None*)

Updates the birth date of a specific resident.

This method searches for a resident by their unique ID and updates their *birthDate* field with the provided new birth date.

Parameters

- **idResident** (*int*) – The unique identifier of the resident whose birth date will be updated.
- **new_birthDate** (*str*) – The new birth date to assign to the resident in the format 'YYYY-MM-DD'.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "message": "Birth date updated successfully"}:

If the birth date is successfully updated. - {"status": "error", "message": <error_message>}: If an error occurs, such as the resident not being found or a database issue.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

Notes

- The method automatically converts the *new_birthDate* string into a *date* object if required.
- Ensure the *new_birthDate* is in the format 'YYYY-MM-DD' to avoid conversion errors.

updateResidentGender(*idResident: int*, *new_gender: str*, *session=None*)

Updates the gender of a specific resident.

This method searches for a resident by their unique ID and updates their *gender* field with the provided new gender. It validates that the new gender is one of the allowed values.

Parameters

- **idResident** (*int*) – The unique identifier of the resident whose gender will be updated.
- **new_gender** (*str*) – The new gender to assign to the resident. Accepted values are "M", "F", or "Otro".
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "message": "Gender updated successfully"}:

If the gender is successfully updated. - {"status": "error", "message": "Invalid gender. Allowed values are 'M', 'F', or 'Otro'"}: If the provided gender is not valid. - {"status": "error", "message": <error_message>}: If an error occurs, such as the resident not being found or a database issue.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

Notes

- Valid gender values are "M", "F", or "Otro". Any other value will result in an error.
- Ensure that the new gender conforms to these accepted values to avoid validation errors.

updateResidentName(*idResident: int*, *new_name: str*, *session=None*)

Updates the name of a specific resident.

This method searches for a resident by their unique ID and updates their *name* field with the provided new name.

Parameters

- **idResident** (*int*) – The unique identifier of the resident whose name will be updated.
- **new_name** (*str*) – The new name to assign to the resident.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "message": "Name updated successfully"}:

If the name is successfully updated. - {"status": "error", "message": <error_message>}: If an error occurs, such as the resident not being found or a database issue.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

updateResidentRoom(*resident_id: int*, *new_room_id: int*, *session=None*)

Updates the room ID (*idRoom*) of a specific resident.

This method searches for a resident by their ID and assigns them to a new room by updating their *idRoom* field.

Parameters

- **resident_id** (*int*) – The unique identifier of the resident whose room will be updated.
- **new_room_id** (*int*) – The ID of the new room to assign to the resident.

- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "message": "Room updated successfully."}:

If the room is updated successfully. - {"status": "error", "message": <error_message>}: If an error occurs, such as the resident not being found or a database issue.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

updateResidentSurname(*idResident: int*, *new_surname: str*, *session=None*)

Updates the surname of a specific resident.

This method searches for a resident by their unique ID and updates their *surname* field with the provided new surname.

Parameters

- **idResident** (*int*) – The unique identifier of the resident whose surname will be updated.
- **new_surname** (*str*) – The new surname to assign to the resident.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "message": "Surname updated successfully"}:

If the surname is successfully updated. - {"status": "error", "message": <error_message>}: If an error occurs, such as the resident not being found or a database issue.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

update_resident(*idResident: int*, *updates: dict*, *session=None*)

Updates an existing resident's details in the database.

This method searches for a resident by their unique ID and applies the specified updates to the resident's record.

Parameters

- **idResident** (*int*) – The unique identifier of the resident to update.
- **updates** (*dict*) – A dictionary containing the fields to update and their new values. .. rubric:: Example

{
“name”: “Jane”, “surname”: “Smith”, “idRoom”: 102, “update”: date(2024, 11, 14)
}

- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”}:

If the update is successful. - {“status”: “not found”}: If no resident with the given ID is found. - {“status”: “error”, “message”: <error_message>}: If an error occurs during the operation.

Return type
dict

Raises
Exception – If an unexpected error occurs during the update.

Process:

1. Check if a database session is provided; if not, create a new session.
2. Query the database for a resident matching the provided *idResident*.
3. **If the resident is found:**
   - Update the specified fields with the provided values.
   - Commit the changes to the database.
   - Return a success status.
4. If the resident is not found, return a “not found” status.

1.7 room_controller module

class app.controllers.room_controller.**RoomController**(*db_url=None*)

Bases: object

access_room(*idResident: int*, *idRoom: int*, *session=None*)

Determines if a resident can access a specific room based on the room’s capacity, type, and family assignment.

This method checks the access rules for a resident attempting to enter a room: - Verifies if the resident and room exist. - Denies access to restricted rooms like “Maintenance.” - Ensures the room is not over capacity. - Allows access to public or common rooms. - Grants access if the room belongs to the resident’s family. - Denies access otherwise.

Parameters

- **idResident** (*int*) – The unique identifier of the resident attempting to access the room.
- **idRoom** (*int*) – The unique identifier of the room the resident is trying to access.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

A message indicating the result of the access attempt:

- ”Resident not found.”: If the resident does not exist.
- ”Room not found.”: If the room does not exist.
- ”Access denied. No puedes entrar a la sala de mantenimiento.”: If the room is restricted (e.g., Maintenance).
- ”Access denied. La sala está llena.”: If the room is at maximum capacity.
- ”Access granted. Welcome to the room.”: If access is granted to public or family-assigned rooms.
- ”Access denied. You are in the wrong room.”: If the resident is not assigned to the room.

Return type
str

Raises
Exception – If an unexpected error occurs during the operation.

Notes

- Public or common rooms (not starting with “Room”) are accessible to all residents.
- Family-assigned rooms are only accessible to members of the assigned family.
- The room’s current occupancy is checked against its maximum capacity.

create_room(*body:* Room, *session=None*)

Creates a new room in the database.

This method validates the existence of the admin and shelter before creating a room. It also ensures that no duplicate rooms exist in the specified shelter.

Parameters

- **body** (*RoomModel*) – The data of the room to be created, including: - idRoom (int, optional): The unique identifier for the room. - roomName (str): The name of the room. - createdBy (int): The ID of the admin who creates the room. - createDate (date): The date the room is created. - idShelter (int): The ID of the shelter where the room belongs. - maxPeople (int): The maximum capacity of the room.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”}:

If the room is successfully created. - {“status”: “error”, “message”: “The admin does not exist”}: If the provided admin ID does not exist in the database. - {“status”: “error”, “message”: “The shelter does not exist”}: If the provided shelter ID does not exist in the database. - {“status”: “error”, “message”: “The room already exists in this shelter”}: If a room with the same name already exists in the specified shelter. - {“status”: “error”, “message”: <error_message>}: If any other error occurs during the operation.

Return type
dict

Raises
Exception – If an unexpected error occurs during the operation.

Notes

- Ensure the provided *roomName* is unique within the specified shelter.
- The *createdBy* field must correspond to an existing admin, and *idShelter* must correspond to an existing shelter.

list_rooms(*session=None*)

Lists all rooms with basic information.

This method retrieves a list of all rooms in the database, including their basic details such as ID, name, capacity, associated shelter, and creation date.

Parameters
session (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

A list of dictionaries, each representing a room with the following details:

- idRoom (int): The unique identifier of the room.
- roomName (str): The name of the room.
- maxPeople (int): The maximum capacity of the room.
- idShelter (int): The ID of the shelter the room belongs to.
- createDate (str, optional): The creation date of the room in ISO 8601 format.

dict: If an error occurs, a dictionary with:

- {“status”: “error”, “message”: <error_message>}

Return type
list

Raises
Exception – If an unexpected error occurs during the operation.

Example Response:

[

{

“idRoom”: 1, “roomName”: “Room A”, “maxPeople”: 10, “idShelter”: 1, “createDate”: “2024-12-01”

}, {

“idRoom”: 2, “roomName”: “Room B”, “maxPeople”: 8, “idShelter”: 2, “createDate”: “2024-12-02”

}

]

Notes

- The creation date (*createDate*) will be formatted in ISO 8601. If not available, it will be *None*.

list_rooms_Room(*session=None*)

Lists all rooms whose names start with "Room".

This method retrieves a list of rooms from the database where the *roomName* field begins with the string "Room".

Parameters

session (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

A list of dictionaries, each representing a room with the following details:

- idRoom (int): The unique identifier of the room.
- roomName (str): The name of the room.
- maxPeople (int): The maximum capacity of the room.
- idShelter (int): The ID of the shelter the room belongs to.
- createDate (str, optional): The creation date of the room in ISO 8601 format.

dict: If an error occurs, a dictionary with:

- {"status": "error", "message": <error_message>}

Return type

list

Raises

Exception – If an unexpected error occurs during the operation.

Example Response:

[

{

"idRoom": 1, "roomName": "Room A", "maxPeople": 10, "idShelter": 1, "createDate": "2024-12-01"

}, {

"idRoom": 2, "roomName": "Room B", "maxPeople": 8, "idShelter": 1, "createDate": "2024-12-02"

}

]

Notes

- Only rooms whose *roomName* starts with "Room" will be included in the response.
- The creation date (*createDate*) will be formatted in ISO 8601. If not available, it will be *None*.

list_rooms_with_resident_count(*session=None*)

Lists all rooms along with the count of residents in each room.

This method retrieves a list of all rooms in the database, including their details and the number of residents currently assigned to each room.

Parameters

session (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

A list of dictionaries, each representing a room and its resident count.

Each dictionary contains:

- idRoom (int): The unique identifier of the room.
- roomName (str): The name of the room.
- maxPeople (int): The maximum capacity of the room.
- resident_count (int): The number of residents currently in the room.

dict: If an error occurs, a dictionary with:

- {"status": "error", "message": <error_message>}

Return type

list

Raises

Exception – If an unexpected error occurs during the operation.

Example Response:

[

{

"idRoom": 1, "roomName": "Room A", "maxPeople": 10, "resident_count": 5

}, {

"idRoom": 2, "roomName": "Room B", "maxPeople": 8, "resident_count": 0

}

]

Notes

- Rooms without residents will still appear in the list with a *resident_count* of 0.
- The method uses an *outerjoin* to include rooms without any assigned residents.

updateRoomName(*idRoom: int*, *new_name: str*, *session=None*)

Updates the name of a specific room.

This method searches for a room by its unique ID and updates its *roomName* field with the provided new name.

Parameters

- **idRoom** (*int*) – The unique identifier of the room whose name will be updated.
- **new_name** (*str*) – The new name to assign to the room.

- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “message”: “Room name updated successfully”}:

If the room name is successfully updated. - {“status”: “error”, “message”: “Room not found”}: If no room with the given ID is found in the database. - {“status”: “error”, “message”: <error_message>}: If an error occurs during the operation.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

1.8 shelter_controller module

class app.controllers.shelter_controller.**ShelterController**(*db_url=None*)

Bases: object

get_shelter_energy_level(*session=None*)

Retrieves the energy level of the first shelter in the database.

This method fetches the energy level of the first shelter record found in the database. If no shelter is found, an error is raised.

Parameters
session (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

A dictionary containing the energy level of the shelter:

- {“energyLevel”: <energy_level>}:

Where <*energy_level*> is the current energy level of the shelter.

Return type
dict

Raises

- **ValueError** – If no shelter record is found in the database.
- **Exception** – If an unexpected error occurs during the operation.

Example Response:
{“energyLevel”: 85}

Notes

- This method assumes there is only one shelter or that the first shelter

record is the one of interest.

get_shelter_radiation_level(*session=None*)

Retrieves the radiation level of the first shelter in the database.

This method fetches the radiation level of the first shelter record found in the database. If no shelter is found, an error is raised.

Parameters

session (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

A dictionary containing the radiation level of the shelter:

- {"radiationLevel": <radiation_level>}:

Where <*radiation_level*> is the current radiation level of the shelter.

Return type

dict

Raises

- **ValueError** – If no shelter record is found in the database.
- **Exception** – If an unexpected error occurs during the operation.

Example Response:

{"radiationLevel": 15}

Notes

- This method assumes there is only one shelter or that the first shelter

record is the one of interest.

get_shelter_water_level(*session=None*)

Retrieves the water level of the first shelter in the database.

This method fetches the water level of the first shelter record found in the database. If no shelter is found, an error is raised.

Parameters

session (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

A dictionary containing the water level of the shelter:

- {"waterLevel": <water_level>}:

Where <*water_level*> is the current water level of the shelter.

Return type

dict

Raises

- **ValueError** – If no shelter record is found in the database.
- **Exception** – If an unexpected error occurs during the operation.

Example Response:
{“waterLevel”: 70}

Notes

- This method assumes there is only one shelter or that the first shelter

record is the one of interest.

updateShelterEnergyLevel(*new_energy_level: int*, *session=None*)

Updates the energy level of the only available shelter.

This method retrieves the shelter record with a fixed ID (assumed to be 1) and updates its energy level to the provided value.

Parameters

- **new_energy_level** (*int*) – The new energy level to assign to the shelter.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {“status”: “ok”, “message”: “Energy level updated successfully”}:

If the energy level is successfully updated. - {“status”: “error”, “message”: “Shelter not found”}: If no shelter with the given ID is found in the database. - {“status”: “error”, “message”: <error_message>}: If an error occurs during the operation.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

Notes

- This method assumes there is only one shelter in the system with *idShelter = 1*.
- Ensure that the *new_energy_level* value is within the acceptable range defined by your application logic.

Example Response:
{“status”: “ok”, “message”: “Energy level updated successfully”}

updateShelterRadiationLevel(*new_radiation_level: int*, *session=None*)

Updates the radiation level of the only available shelter.

This method retrieves the shelter record with a fixed ID (assumed to be 1) and updates its radiation level to the provided value.

Parameters

- **new_radiation_level** (*int*) – The new radiation level to assign to the shelter.

- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "message": "Radiation level updated successfully"}:

If the radiation level is successfully updated. - {"status": "error", "message": "Shelter not found"}: If no shelter with the given ID is found in the database. - {"status": "error", "message": <error_message>}: If an error occurs during the operation.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

Notes

- This method assumes there is only one shelter in the system with *idShelter = 1*.
- Ensure that the *new_radiation_level* value is within the acceptable range defined by your application logic.

Example Response:
{"status": "ok", "message": "Radiation level updated successfully"}

updateShelterWaterLevel(*new_water_level: int*, *session=None*)

Updates the water level of the only available shelter.

This method retrieves the shelter record with a fixed ID (assumed to be 1) and updates its water level to the provided value.

Parameters

- **new_water_level** (*int*) – The new water level to assign to the shelter.
- **session** (*Session, optional*) – SQLAlchemy session object for database interaction. If not provided, a new session will be created.

Returns

Result of the operation.

- {"status": "ok", "message": "Water level updated successfully"}:

If the water level is successfully updated. - {"status": "error", "message": "Shelter not found"}: If no shelter with the given ID is found in the database. - {"status": "error", "message": <error_message>}: If an error occurs during the operation.

Return type
dict

Raises

- **SQLAlchemyError** – If a database-related error occurs.
- **Exception** – If an unexpected error occurs during the operation.

Notes

- This method assumes there is only one shelter in the system with *idShelter = 1*.
- Ensure that the *new_water_level* value is within the acceptable range defined by your application logic.

Example Response:

{“status”: “ok”, “message”: “Water level updated successfully”}

CHAPTER

TWO

MODELS PACKAGE

2.1 admin module

class app.models.admin.**Admin**(*, *idAdmin: int | None = None*, *email: str*, *name: str*, *password: str*)

Bases: BaseModel

Represents an admin within the shelter system, including credentials and contact information.

idAdmin

Unique identifier for the admin.

Type
int

email

Contact email of the admin.

Type
str

name

Name of the admin.

Type
str

password

Password for admin authentication.

Type
str

2.2 alarm module

class app.models.alarm.**Alarm**(*, *idAlarm: int | None = None*, *start: datetime*, *end: datetime | None = None*, *idRoom: int*, *createDate: datetime*)

Bases: BaseModel

Represents an alarm event within the shelter system, logging the start and end times, and associating the alarm with specific room, resident, and admin IDs.

idAlarm

Unique identifier for the alarm.

Type
int

start

The timestamp when the alarm started.

Type
datetime

end

The timestamp when the alarm ended.

Type
datetime

idRoom

Foreign key referring to the room associated with the alarm.

Type
int

idResident

Foreign key referring to the resident associated with the alarm.

Type
int

idAdmin

Foreign key referring to the admin responsible for the alarm.

Type
int

createDate

The timestamp when the alarm record was created.

Type
datetime

2.3 family module

class app.models.family.**Family**(*, *idFamily: int | None = None*, *familyName: str*, *idRoom: int*, *idShelter: int*, *createdBy: int*, *createDate: date*)

Bases: BaseModel

Represents a family residing in the shelter, with information about their assigned room and shelter.

idFamily

Unique identifier for the family.

Type
int

familyName

The name or identifier of the family.

Type
str

idRoom

Foreign key to the room assigned to the family.

Type
int

idShelter

Foreign key to the shelter where the family resides.

Type
int

createdBy

ID of the admin who created this family record.

Type
int

createDate

The date when the family record was created.

Type
date

2.4 machine module

class app.models.machine.**Machine**(*, *idMachine: int*, *machineName: str*, *on: bool*, *idRoom: int*, *createdBy: int*, *createDate: date*, *update: date | None = None*)

Bases: BaseModel

Represents a machine in the shelter, including its operational status and location.

idMachine

Unique identifier for the machine.

Type
int

machineName

Name or identifier of the machine.

Type
str

on

Operational status of the machine (True if on, False if off).

Type
bool

idRoom

Foreign key to the room where the machine is located.

Type
int

createdBy

ID of the admin who created this machine record.

Type
int

createDate

The date when the machine record was created.

Type
date

update

The date when the machine record was last updated.

Type
Optional[date]

2.5 resident module

class app.models.resident.**Resident**(*, *idResident: int | None = None*, *name: str*, *surname: str*, *birthDate: date*, *gender: str*, *createdBy: int*, *createDate: date*, *update: date | None = None*, *idFamily: int | None = None*, *idRoom: int | None = None*)

Bases: BaseModel

Represents a resident within the shelter system, including personal information and family association.

idResident

Unique identifier for the resident.

Type
int

name

First name of the resident.

Type
str

surname

Surname of the resident.

Type
str

birthDate

Date of birth of the resident.

Type
date

gender

Gender of the resident.

Type
str

createdBy

ID of the admin who created this resident record.

Type
int

createDate

The date when the resident record was created.

Type
date

update

The date when the resident record was last updated.

Type
Optional[date]

idFamily

Foreign key to the family the resident is associated with.

Type
Optional[int]

idRoom

Foreign key to the room the resident is.

Type
int | None

2.6 room module

class app.models.room.**Room**(*, *idRoom: int | None = None*, *roomName: str*, *createdBy: int*, *createDate: date*, *idShelter: int*, *maxPeople: int*)

Bases: BaseModel

Represents a room within a shelter, with information about capacity and creation.

idRoom

Unique identifier for the room.

Type
int

roomName

Name or identifier of the room.

Type
str

createdBy

ID of the admin who created this room record.

Type
int

createDate

The date when the room record was created.

Type
date

idShelter

Foreign key to the shelter where the room is located.

Type
int

maxPeople

Maximum number of people that can occupy the room.

Type
int

2.7 shelter module

class app.models.shelter.**Shelter**(*, *idShelter: int*, *shelterName: str*, *address: str*, *phone: str | None = None*, *email: str | None = None*, *maxPeople: int*, *energyLevel: int*, *waterLevel: int*, *radiationLevel: int*)

Bases: BaseModel

Represents a shelter with capacity and contact information.

idShelter

Unique identifier for the shelter.

Type
int

shelterName

Name of the shelter.

Type
str

address

Physical address of the shelter.

Type
str

phone

Contact phone number for the shelter.

Type
Optional[str]

email

Contact email address for the shelter.

Type
Optional[str]

maxPeople

Maximum number of people the shelter can accommodate.

Type
int

energyLevel

Energy level of the shelter.

Type
int

waterLevel

Water level of the shelter.

Type
int

radiationLevel

Water level of the shelter.

Type
int

CHAPTER

THREE

MYSQL PACKAGE

3.1 admin module

class app.mysql.admin.**Admin**(**kwargs)

Bases: Base

Represents an Admin entity in the system.

idAdmin

The unique identifier for the Admin.

Type
int

email

The email of the Admin.

Type
str

name

The name of the Admin.

Type
str

password

The hashed password of the Admin.

Type
str

3.2 alarm module

class app.mysql.alarm.**Alarm**(**kwargs)

Bases: Base

Represents an alarm entity in the system.

The alarm tracks events occurring within a room and associates them with a resident and an admin. It includes a start and end time for the event, as well as metadata for auditing.

idAlarm

Unique identifier for the alarm.

Type
int

start
Timestamp indicating when the alarm starts.

Type
datetime

end
Timestamp indicating when the alarm ends.

Type
datetime

idRoom
Foreign key linking the alarm to a specific room.

Type
int

idResident
Foreign key linking the alarm to a specific resident.

Type
int

idAdmin
Foreign key linking the alarm to the admin responsible.

Type
int

createDate
Timestamp indicating when the alarm was created.

Type
datetime

3.3 base module

3.4 family module

class app.mysql.family.**Family**(**kwargs*)

Bases: Base

Represents a family entity in the system.

This entity links a family to a specific room and shelter, and tracks metadata about its creation.

idFamily
Unique identifier for the family.

Type
int

familyName
The name of the family.

Type
str

idRoom

Foreign key linking the family to a specific room.

Type
int

idShelter

Foreign key linking the family to a specific shelter.

Type
int

createdBy

The ID of the admin or user who created the family entry.

Type
int

createDate

The date when the family entry was created.

Type
date

3.5 initializeData module

app.mysql.initializeData.**initialize_database**()

Seeds the database with initial data for all entities if the database is empty. This function ensures that the database is initialized with a default set of data for all core entities. It checks whether each entity's table is empty before populating it with predefined data. This is useful for setting up the application with consistent initial data when running for the first time.

Entities covered:

- Admin: Represents administrative users of the system.
- Family: Represents families residing in the shelter.
- Machine: Represents machines associated with rooms in the shelter.
- Resident: Represents individual residents in the shelter.
- Room: Represents rooms available in the shelter.
- Shelter: Represents the shelter itself, including metadata such as capacity and resource levels.

Data Seeded:

- Admins: Two sample admin accounts with email, name, and password.
- Shelter: A single shelter with attributes like address, phone, energy level, etc.
- Rooms: A set of rooms with details about their capacity and assignment to the shelter.
- Families: A couple of families assigned to specific rooms in the shelter.
- Residents: Sample residents, some assigned to families and others not.
- Machines: A machine (e.g., a heater) associated with a room.

Process:

1. Checks if each entity's table is empty.
2. If empty, inserts predefined data for that entity.
3. Commits the session to persist the data.
4. Handles errors by rolling back the session and printing the error message.

3.6 machine module

class app.mysql.machine.**Machine**(***kwargs*)

Bases: Base

Represents a machine entity in the system.

This entity tracks details about a specific machine, its status, and its association with a room. It also includes metadata for creation and updates.

idMachine

Unique identifier for the machine.

Type
int

machineName

The name of the machine.

Type
str

on

Indicates whether the machine is currently on (True) or off (False).

Type
bool

idRoom

Foreign key linking the machine to a specific room.

Type
int

createdBy

The ID of the admin or user who created the machine entry.

Type
int

createDate

The date when the machine entry was created.

Type
date

update

The date when the machine entry was last updated.

Type
date

3.7 mysql module

3.8 resident module

class app.mysql.resident.**Resident**(***kwargs*)

Bases: Base

Represents a resident entity in the system.

This entity stores information about a resident, including personal details, family association, and metadata for auditing purposes.

idResident

Unique identifier for the resident.

Type
int

name

First name of the resident.

Type
str

surname

Last name of the resident.

Type
str

birthDate

The birthdate of the resident.

Type
date

gender

The gender of the resident.

Type
str

createdBy

The ID of the admin or user who created the resident entry.

Type
int

createDate

The date when the resident entry was created.

Type
date

update

The date when the resident entry was last updated.

Type
date

idFamily

Foreign key linking the resident to a specific family.

Type
int

idRoom

Foreign key linking the resident to a specific room.

Type
int

3.9 room module

class app.mysql.room.**Room**(**kwargs*)

Bases: Base

Represents a room entity in the system.

This entity stores information about a room, including its name, the shelter it belongs to, and its capacity. It also includes metadata for creation and tracking purposes.

idRoom

Unique identifier for the room.

Type
int

roomName

The name of the room.

Type
str

createdBy

The ID of the admin or user who created the room entry.

Type
int

createDate

The date when the room entry was created.

Type
date

idShelter

Foreign key linking the room to a specific shelter.

Type
int

maxPeople

The maximum number of people the room can accommodate.

Type
int

3.10 shelter module

class app.mysql.shelter.**Shelter**(***kwargs*)

Bases: Base

Represents a shelter entity in the system.

This entity stores information about a shelter, including its name, address, contact information, and maximum capacity.

idShelter

Unique identifier for the shelter.

Type
int

shelterName

The name of the shelter.

Type
str

address

The address of the shelter.

Type
str

phone

Contact phone number for the shelter.

Type
str

email

Contact email for the shelter.

Type
str

maxPeople

The maximum number of people the shelter can accommodate.

Type
int

energyLevel

The energy leverl of the shelter.

Type
int

waterLevel

The water level of the shelter.

Type
int

radiationLevel

The radiation level of the shelter.

Type
int

CHAPTER

FOUR

TEST PACKAGE

4.1 conftest module

test.conftest.**setup_database**()

Fixture to create an SQLite in-memory database for testing.

4.2 test_admin module

test.test_admin.**test_create_admin_database_error**(*setup_database*, *mocker*)

Test: Handle a database error during admin creation.

Steps:

1. Simulate a database error by mocking the session's commit method.
2. Attempt to create a new admin.
3. Verify the response indicates a database error.

Expected Outcome:

- The operation should return an error response with the exception message.
- No admin should be added to the database.

Parameters

- **setup_database** (*fixture*) – The database session used for test setup.
- **mocker** (*pytest-mock*) – Mocking library for simulating errors.

test.test_admin.**test_create_admin_duplicate_email**(*setup_database*)

Test: Prevent creating an admin with a duplicate email.

Steps:

1. Add an admin with a specific email to the database.
2. Attempt to create another admin with the same email.
3. Verify the response indicates an error.

Expected Outcome:

- The second attempt should fail with a duplicate email error.
- The database should only contain one admin with the given email.

Parameters
setup_database (*fixture*) – The database session used for test setup.

test.test_admin.**test_create_admin_success**(*setup_database*)

Test: Successfully create a new admin.

Steps:

1. Create a new admin with valid details.
2. Verify the response is successful.
3. Confirm the admin is added to the database with the correct details.

Expected Outcome:

- The admin should be successfully created.
- The admin should exist in the database with the provided details.

Parameters
setup_database (*fixture*) – The database session used for test setup.

test.test_admin.**test_delete_admin_not_found**(*setup_database*)

Test: Verify that the method handles the case when the admin ID is not found.

Steps:

1. Ensure the database does not contain an admin with the specified ID.
2. Call the *deleteAdmin* method with a non-existent ID.
3. Verify the response indicates that the admin was not found.

Expected Outcome:

- The method returns a status of “error”.
- The response contains the message “Admin not found”.

test.test_admin.**test_delete_admin_success**(*setup_database*)

Test: Verify that an admin is successfully deleted by their ID.

Steps:

1. Add an admin to the database with a known ID.
2. Call the *deleteAdmin* method with the admin’s ID.
3. Verify the admin is removed from the database.

Expected Outcome:

- The method returns a status of “ok”.
- The admin record is no longer present in the database.

test.test_admin.**test_delete_admin_unexpected_error**(*setup_database*, *mocker*)

Test: Verify that the method handles unexpected errors gracefully.

Steps:

1. Mock the database session’s *delete* method to raise an exception.
2. Call the *deleteAdmin* method.

3. Verify the response indicates an error occurred.

Expected Outcome:

- The method returns a status of “error”.
- The response contains the exception message.

test.test_admin.**test_get_admin_by_id_not_found**(*setup_database*)

Test: Verify that the method returns an error when the admin ID is not found.

Steps:

1. Ensure the database does not contain an admin with the specified ID.
2. Call the *getAdminById* method with a non-existent ID.
3. Verify the response indicates that the admin was not found.

Expected Outcome:

- The method returns a status of “error”.
- The response contains the message “Administrador no encontrado”.

test.test_admin.**test_get_admin_by_id_success**(*setup_database*)

Test: Verify that the method retrieves an admin successfully by their ID.

Steps:

1. Add an admin to the database with a known ID.
2. Call the *getAdminById* method with the admin’s ID.
3. Verify the response contains the correct admin details.

Expected Outcome:

- The method returns a status of “ok”.
- The admin details in the response match the database record.

test.test_admin.**test_get_admin_by_id_unexpected_error**(*setup_database*, *mocker*)

Test: Verify that the method handles unexpected errors gracefully.

Steps:

1. Mock the database query to raise an exception.
2. Call the *getAdminById* method.
3. Verify the response indicates an error occurred.

Expected Outcome:

- The method returns a status of “error”.
- The response contains the exception message.

test.test_admin.**test_list_admins_empty**(*setup_database*)

Test: Verify that the method handles an empty database gracefully.

Steps:

1. Ensure no admins are present in the database.
2. Call the *listAdmins* method.
3. Verify the response contains an empty list.

Expected Outcome:

- The method returns a status of “ok”.
- The response contains an empty list of admins.

test.test_admin.**test_list_admins_success**(*setup_database*)

Test: Verify that the method retrieves all admins successfully.

Steps:

1. Add multiple admins to the database.
2. Call the *listAdmins* method.
3. Verify the response contains all the admins.

Expected Outcome:

- The method returns a status of “ok”.
- The response contains a list of all admins with their IDs and emails.

test.test_admin.**test_list_admins_unexpected_error**(*setup_database*, *mocker*)

Test: Verify that the method handles unexpected errors gracefully.

Steps:

1. Mock the database session’s *query* method to raise an exception.
2. Call the *listAdmins* method.
3. Verify the response indicates an error occurred.

Expected Outcome:

- The method returns a status of “error”.
- The response contains the exception message.

test.test_admin.**test_login_admin_invalid_email**(*setup_database*)

Test: Verify that the method handles incorrect email credentials.

Steps:

1. Add an admin to the database with a known email and password.
2. Call the *loginAdmin* method with an incorrect email.
3. Verify the response indicates invalid credentials.

Expected Outcome:

- The method returns a status of “error”.
- The response contains the message “Invalid credentials”.

test.test_admin.**test_login_admin_invalid_password**(*setup_database*)

Test: Verify that the method handles incorrect password credentials.

Steps:

1. Add an admin to the database with a known email and password.
2. Call the *loginAdmin* method with an incorrect password.
3. Verify the response indicates invalid credentials.

Expected Outcome:

- The method returns a status of “error”.
- The response contains the message “Invalid credentials”.

test.test_admin.**test_login_admin_nonexistent_user**(*setup_database*)

Test: Verify that the method handles a login attempt for a non-existent admin.

Steps:

1. Ensure the database does not contain any admins.
2. Call the *loginAdmin* method with any credentials.
3. Verify the response indicates invalid credentials.

Expected Outcome:

- The method returns a status of “error”.
- The response contains the message “Invalid credentials”.

test.test_admin.**test_login_admin_success**(*setup_database*)

Test: Verify that the method successfully logs in an admin with correct credentials.

Steps:

1. Add an admin to the database with known email and password.
2. Call the *loginAdmin* method with valid credentials.
3. Verify the response contains a JWT token and user information.

Expected Outcome:

- The method returns a status of “ok”.
- A valid JWT token is included in the response.
- User information (ID and email) is correct.

test.test_admin.**test_login_admin_unexpected_error**(*setup_database*, *mocker*)

Test: Verify that the method handles unexpected errors gracefully.

Steps:

1. Mock the database query to raise an exception.
2. Call the *loginAdmin* method.
3. Verify the response indicates an error occurred.

Expected Outcome:

- The method returns a status of “error”.
- The response contains the exception message.

test.test_admin.**test_update_admin_email_database_error**(*setup_database*, *mocker*)

Test: Verify that the method handles database errors gracefully.

Steps:

1. Mock the database session’s commit method to raise an SQLAlchemyError.
2. Call the *updateAdminEmail* method.
3. Verify the response indicates a database error.

Expected Outcome:

- The method returns a status of “error”.
- The response contains an error message indicating a database issue.

test.test_admin.**test_update_admin_email_not_found**(*setup_database*)

Test: Verify that the method handles the case where the admin is not found.

Steps:

1. Ensure no admins exist in the database.
2. Call the *updateAdminEmail* method with a non-existent admin ID.
3. Verify the response indicates that the admin is not found.

Expected Outcome:

- The method returns a status of “error”.
- The response contains a “Administrador no encontrado” message.

test.test_admin.**test_update_admin_email_success**(*setup_database*)

Test: Verify that the method successfully updates the email of an admin.

Steps:

1. Add an admin to the database.
2. Call the *updateAdminEmail* method with a new email.
3. Verify the response indicates success.
4. Confirm that the email is updated in the database.

Expected Outcome:

- The method returns a status of “ok”.
- The admin’s email is updated in the database.

test.test_admin.**test_update_admin_email_unexpected_error**(*setup_database*, *mocker*)

Test: Verify that the method handles unexpected errors gracefully.

Steps:

1. Mock the database session’s *query* method to raise a generic exception.
2. Call the *updateAdminEmail* method.
3. Verify the response indicates an error occurred.

Expected Outcome:

- The method returns a status of “error”.
- The response contains the exception message.

test.test_admin.**test_update_admin_name_database_error**(*setup_database*, *mocker*)

Test: Verify that the method handles database errors gracefully.

Steps:

1. Mock the database session to raise an SQLAlchemyError.
2. Call the *updateAdminName* method.
3. Verify the response indicates a database error.

Expected Outcome:

- The method returns an error message indicating a database issue.

test.test_admin.**test_update_admin_name_not_found**(*setup_database*)

Test: Verify that the method returns an error if the admin is not found.

Steps:

1. Ensure no admin exists in the database.
2. Call the *updateAdminName* method with a non-existent admin ID.
3. Verify the response indicates the admin was not found.

Expected Outcome:

- The method returns an error message indicating the admin was not found.

test.test_admin.**test_update_admin_name_success**(*setup_database*)

Test: Verify that the name of an admin can be updated successfully.

Steps:

1. Add an admin to the database.
2. Call the *updateAdminName* method to update the admin's name.
3. Verify the response and the updated name in the database.

Expected Outcome:

- The method returns a success message.
- The admin's name is updated in the database.

test.test_admin.**test_update_admin_name_unexpected_error**(*setup_database*, *mocker*)

Test: Verify that the method handles unexpected errors gracefully.

Steps:

1. Mock the session to raise a general Exception.
2. Call the *updateAdminName* method.
3. Verify the response indicates an unexpected error.

Expected Outcome:

- The method returns an error message indicating an unexpected issue.

test.test_admin.**test_update_admin_password_database_error**(*setup_database*, *mocker*)

Test: Verify that the method handles database errors gracefully.

Steps:

1. Mock the database session's commit method to raise an SQLAlchemyError.
2. Call the *updateAdminPassword* method.
3. Verify the response indicates a database error.

Expected Outcome:

- The method returns a status of "error".
- The response contains an error message indicating a database issue.

test.test_admin.**test_update_admin_password_not_found**(*setup_database*)

Test: Verify that the method handles the case where the admin is not found.

Steps:

1. Ensure no admins exist in the database.
2. Call the *updateAdminPassword* method with a non-existent admin ID.
3. Verify the response indicates that the admin is not found.

Expected Outcome:

- The method returns a status of "error".
- The response contains a "Administrador no encontrado" message.

test.test_admin.**test_update_admin_password_success**(*setup_database*)

Test: Verify that the method successfully updates the password of an admin.

Steps:

1. Add an admin to the database.
2. Call the *updateAdminPassword* method with a new password.
3. Verify the response indicates success.
4. Confirm that the password is updated in the database.

Expected Outcome:

- The method returns a status of "ok".
- The admin's password is updated in the database.

test.test_admin.**test_update_admin_password_unexpected_error**(*setup_database*, *mocker*)

Test: Verify that the method handles unexpected errors gracefully.

Steps:

1. Mock the database session's *query* method to raise a generic exception.
2. Call the *updateAdminPassword* method.
3. Verify the response indicates an error occurred.

Expected Outcome:

- The method returns a status of "error".
- The response contains the exception message.

4.3 test_alarm module

test.test_alarm.**test_create_alarm**(*mocker*, *setup_database*)

Test: Validate the functionality of the *create_alarm* method.

Steps:

1. Add a room to the database.
2. Test successful creation of an alarm.
3. Test creation of an alarm for a non-existent room.
4. Test creation of a duplicate alarm.

5. Mock the session commit to raise a SQLAlchemyError and handle it.

Expected Outcome:

- Alarm is created successfully if the room exists and no duplicates are found.
- Error is returned if the room does not exist.
- Error is returned if a duplicate alarm is detected.
- SQLAlchemy errors are handled gracefully.

Parameters

- **mocker** (*pytest fixture*) – Used for mocking objects and methods.
- **setup_database** (*fixture*) – The database session used for test setup.

test.test_alarm.**test_create_alarm_level_database_error**(*setup_database*, *mocker*)

Test: Verify that the method handles database errors gracefully.

Steps:

1. Mock the database session to raise an SQLAlchemyError.
2. Call the *create_alarmLevel* method with valid alarm data.
3. Verify the response indicates a database error.

Expected Outcome:

- The method returns an error status.
- The response contains a message indicating a database error.

test.test_alarm.**test_create_alarm_level_room_not_exist**(*setup_database*)

Test: Verify that the method returns an error if room 3 does not exist.

Steps:

1. Ensure the database does not contain room 3.
2. Call the *create_alarmLevel* method with valid alarm data.
3. Verify that the response indicates the room does not exist.

Expected Outcome:

- The method returns an error status.
- The response contains a message indicating that room 3 does not exist.
- No alarm is created in the database.

test.test_alarm.**test_create_alarm_level_success**(*setup_database*)

Test: Verify that the method successfully creates an alarm in room 3.

Steps:

1. Add room 3 to the database.
2. Call the *create_alarmLevel* method with valid alarm data.
3. Verify that the alarm is created and the response contains the correct details.

Expected Outcome:

- The method returns a success status.

- The response includes the generated *idAlarm* and a success message.
- The database contains the new alarm.

test.test_alarm.**test_list_alarms_database_error**(*setup_database*, *mocker*)

Test: Verify that the method handles database errors gracefully.

Steps:

1. Mock the session query to raise an exception.
2. Call the *list_alarms* method.
3. Verify the response indicates a database error.

Expected Outcome:

- The method returns an error status.
- The response contains an error message indicating a database issue.

test.test_alarm.**test_list_alarms_empty_database**(*setup_database*)

Test: Verify that the method handles an empty database gracefully.

Steps:

1. Ensure the database has no alarms.
2. Call the *list_alarms* method.
3. Verify the response indicates no alarms are found.

Expected Outcome:

- The method returns a success status.
- The response contains an empty list of alarms.

test.test_alarm.**test_list_alarms_success**(*setup_database*)

Test: Verify that the method retrieves all alarms in the database successfully.

Steps:

1. Add multiple alarms to the database.
2. Call the *list_alarms* method.
3. Verify the response contains the correct list of alarms.

Expected Outcome:

- The method returns a success status.
- The response contains all the alarms with their correct details.

test.test_alarm.**test_update_alarm_end_date_database_error**(*setup_database*, *mocker*)

Test: Verify that the method handles database errors gracefully.

Steps:

1. Mock the database session to raise an SQLAlchemyError.
2. Call the *updateAlarmEndDate* method with a valid *idAlarm* and new end date.
3. Verify the response indicates a database error.

Expected Outcome:

- The method returns an error status.

- The response contains a message indicating a database error.

test.test_alarm.**test_update_alarm_end_date_not_found**(*setup_database*)

Test: Verify that the method returns an error if the alarm does not exist.

Steps:

1. Ensure the database does not contain the specified alarm ID.
2. Call the *updateAlarmEndDate* method with a non-existent *idAlarm*.
3. Verify that the response indicates the alarm was not found.

Expected Outcome:

- The method returns an error status.
- The response contains a message indicating the alarm was not found.

test.test_alarm.**test_update_alarm_end_date_success**(*setup_database*)

Test: Verify that the method updates the end date of an alarm successfully.

Steps:

1. Add an alarm to the database.
2. Call the *updateAlarmEndDate* method with a valid *idAlarm* and new end date.
3. Verify that the end date of the alarm is updated.

Expected Outcome:

- The method returns a success status.
- The alarm's end date is updated in the database.

4.4 test_family module

test.test_family.**test_create_family_duplicate**(*setup_database*)

Test: Prevent creating a duplicate family in the same room.

Steps:

1. Add a room, shelter, admin, and an existing family to the database.
2. Call the *create_family* method with the same family name and room ID.
3. Verify the response indicates that the family already exists.

Expected Outcome:

- The operation should fail with an error message about the duplicate family.
- No duplicate family should be added to the database.

Parameters

setup_database (*fixture*) – The database session used for test setup.

test.test_family.**test_create_family_room_not_exist**(*setup_database*)

Test: Prevent creating a family when the room does not exist.

Steps:

1. Call the *create_family* method with a nonexistent room ID.

2. Verify the response indicates that the room does not exist.

Expected Outcome:

- The operation should fail with an error message about the room.
- No family should be added to the database.

Parameters
setup_database (*fixture*) – The database session used for test setup.

test.test_family.**test_create_family_shelter_not_exist**(*setup_database*)

Test: Prevent creating a family when the shelter does not exist.

Steps:

1. Add a room and admin to the database.
2. Call the *create_family* method with a nonexistent shelter ID.
3. Verify the response indicates that the shelter does not exist.

Expected Outcome:

- The operation should fail with an error message about the shelter.
- No family should be added to the database.

Parameters
setup_database (*fixture*) – The database session used for test setup.

test.test_family.**test_create_family_success**(*setup_database*)

Test: Successfully create a new family.

Steps:

1. Add a room, shelter, and admin to the database.
2. Call the *create_family* method with valid data.
3. Verify the response is successful.
4. Confirm the family is added to the database with the correct details.

Expected Outcome:

- The family should be successfully created.
- The family should exist in the database with the provided details.

Parameters
setup_database (*fixture*) – The database session used for test setup.

test.test_family.**test_delete_family_sqlalchemy_error**(*mocker*, *setup_database*)

Test: Validate handling of SQLAlchemyError during deletion.

test.test_family.**test_delete_family_successful**(*mocker*, *setup_database*)

Test: Validate successful deletion of a family with no members.

test.test_family.**test_delete_family_with_members**(*mocker*, *setup_database*)

Test: Validate error when attempting to delete a family with associated members.

test.test_family.**test_delete_nonexistent_family**(*mocker*, *setup_database*)

Test: Validate error when attempting to delete a family that does not exist.

test.test_family.**test_list_families_empty**(*setup_database*)

Test: Validate behavior when no families are present in the database.

Steps:

1. Ensure the database is empty.
2. Call the *listFamilies* method.

Expected Outcome:

- The method returns a status of "ok" with an empty list.

test.test_family.**test_list_families_sqlalchemy_error**(*mocker*, *setup_database*)

Test: Validate behavior when a SQLAlchemyError occurs during the query.

Steps:

1. Mock the session to raise a SQLAlchemyError during query execution.
2. Call the *listFamilies* method.

Expected Outcome:

- The method returns a status of "error" with the error message.

test.test_family.**test_list_families_success**(*setup_database*)

Test: Validate successful retrieval of families from the database.

Steps:

1. Add multiple families to the database.
2. Call the *listFamilies* method.

Expected Outcome:

- The method returns a status of "ok" and a list of families with correct details.

test.test_family.**test_list_families_unexpected_exception**(*mocker*, *setup_database*)

Test: Validate behavior when an unexpected exception occurs.

Steps:

1. Mock the session to raise a generic exception during query execution.
2. Call the *listFamilies* method.

Expected Outcome:

- The method returns a status of "error" with the exception message.

4.5 test_machine module

test.test_machine.**test_create_machine_admin_not_exist**(*setup_database*)

Test: Prevent creating a machine when the admin does not exist.

Steps:

1. Add a room to the database.
2. Call the *create_machine* method with a nonexistent admin ID.

3. Verify the response indicates that the admin does not exist.

Expected Outcome:

- The operation should fail with an error message about the admin.
- No machine should be added to the database.

Parameters
setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_create_machine_duplicate**(*setup_database*)

Test: Prevent creating a duplicate machine in the same room.

Steps:

1. Add a room, admin, and an existing machine to the database.
2. Call the *create_machine* method with the same machine name and room ID.
3. Verify the response indicates that the machine already exists.

Expected Outcome:

- The operation should fail with an error message about the duplicate machine.
- No duplicate machine should be added to the database.

Parameters
setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_create_machine_room_not_exist**(*setup_database*)

Test: Prevent creating a machine when the room does not exist.

Steps:

1. Call the *create_machine* method with a nonexistent room ID.
2. Verify the response indicates that the room does not exist.

Expected Outcome:

- The operation should fail with an error message about the room.
- No machine should be added to the database.

Parameters
setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_create_machine_success**(*setup_database*)

Test: Successfully create a new machine.

Steps:

1. Add a room and admin to the database.
2. Call the *create_machine* method with valid data.
3. Verify the response is successful.
4. Confirm the machine is added to the database with the correct details.

Expected Outcome:

- The machine should be successfully created.

- The machine should exist in the database with the provided details.

Parameters

setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_delete_machine_db_error**(*mocker*, *setup_database*)

Test: Handle database error during the machine deletion process.

Steps:

1. Add a machine to the database.
2. Mock the database delete operation to raise an SQLAlchemyError.
3. Call the *deleteMachine* method.
4. Verify the response indicates a database error.

Expected Outcome:

- The response should indicate a database error occurred.

Parameters

- **mocker** (*pytest fixture*) – Used for mocking objects and methods.
- **setup_database** (*fixture*) – The database session used for test setup.

test.test_machine.**test_delete_machine_general_exception**(*mocker*, *setup_database*)

Test: Handle a general exception during the machine deletion process.

Steps:

1. Add a machine to the database.
2. Mock the database delete operation to raise a general exception.
3. Call the *deleteMachine* method.
4. Verify the response indicates a general error.

Expected Outcome:

- The response should indicate a general error occurred.

Parameters

- **mocker** (*pytest fixture*) – Used for mocking objects and methods.
- **setup_database** (*fixture*) – The database session used for test setup.

test.test_machine.**test_delete_machine_not_found**(*setup_database*)

Test: Attempt to delete a non-existent machine from the database.

Steps:

1. Ensure no machine exists with the specified ID.
2. Call the *deleteMachine* method with a non-existent machine ID.
3. Verify the response indicates the machine was not found.

Expected Outcome:

- The response should indicate the machine was not found.

Parameters
setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_delete_machine_success**(*setup_database*)

Test: Successfully delete a machine from the database.

Steps:

1. Add a machine to the database.
2. Call the *deleteMachine* method with the machine ID.
3. Verify the machine is deleted from the database.

Expected Outcome:

- The response should indicate successful deletion.

Parameters
setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_list_all_machines_success**(*setup_database*)

Test: Successfully retrieve all machines from the database.

Steps:

1. Add multiple machines to the database.
2. Call the *list_machines* method.
3. Verify the response contains all machines with correct details.

Expected Outcome:

- The response should include all machines in the database with accurate details.

Parameters
setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_list_machines_db_error**(*mocker*, *setup_database*)

Test: Handle database error during the machine listing process.

Steps:

1. Mock the database query to raise an SQLAlchemyError.
2. Call the *list_machines* method.
3. Verify the response indicates a database error.

Expected Outcome:

- The response should indicate a database error occurred.

Parameters

- **mocker** (*pytest fixture*) – Used for mocking objects and methods.
- **setup_database** (*fixture*) – The database session used for test setup.

test.test_machine.**test_list_machines_general_exception**(*mocker*, *setup_database*)

Test: Handle a general exception during the machine listing process.

Steps:

1. Mock the database query to raise a general exception.
2. Call the *list_machines* method.
3. Verify the response indicates a general error.

Expected Outcome:

- The response should indicate a general error occurred.

Parameters

- **mocker** (*pytest fixture*) – Used for mocking objects and methods.
- **setup_database** (*fixture*) – The database session used for test setup.

test.test_machine.**test_list_no_machines**(*setup_database*)

Test: Retrieve an empty list when no machines exist in the database.

Steps:

1. Ensure the database is empty.
2. Call the *list_machines* method.
3. Verify the response indicates no machines are available.

Expected Outcome:

- The response should indicate no machines were found.

Parameters

setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_date_db_error**(*mocker*, *setup_database*)

Test: Handle database error during the update process.

Steps:

1. Add a machine to the database.
2. Mock the database query to raise an SQLAlchemyError.
3. Call the *updateMachineDate* method.
4. Verify the response indicates a database error.

Expected Outcome:

- The response should indicate a database error occurred.

Parameters

- **mocker** (*pytest fixture*) – Used for mocking objects and methods.
- **setup_database** (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_date_general_exception**(*mocker*, *setup_database*)

Test: Handle a general exception during the update process.

Steps:

1. Add a machine to the database.
2. Mock the database query to raise a general exception.
3. Call the *updateMachineDate* method.
4. Verify the response indicates a general error.

Expected Outcome:

- The response should indicate a general error occurred.

Parameters

- **mocker** (*pytest fixture*) – Used for mocking objects and methods.
- **setup_database** (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_date_not_found**(*setup_database*)

Test: Attempt to update the *update* field of a non-existent machine.

Steps:

1. Ensure the database is empty or does not contain the target machine.
2. Call the *updateMachineDate* method with a non-existent machine name.
3. Verify the response indicates an error.

Expected Outcome:

- The response should indicate the machine was not found.

Parameters

setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_date_success**(*setup_database*)

Test: Successfully update the *update* field of a machine to the current date.

Steps:

1. Add a machine to the database with an initial *update* field value.
2. Call the *updateMachineDate* method with the machine name.
3. Verify the response is successful.
4. Confirm the machine's *update* field is updated to the current date.

Expected Outcome:

- The machine's *update* field is updated to today's date.

Parameters

setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_status_db_error**(*mocker*, *setup_database*)

Test: Handle database error during the update process.

Steps:

1. Add a machine to the database.
2. Mock the database query to raise an SQLAlchemyError.
3. Call the *updateMachineStatusOn* method.
4. Verify the response indicates a database error.

Expected Outcome:

- The response should indicate a database error occurred.

Parameters

- **mocker** (*pytest fixture*) – Used for mocking objects and methods.
- **setup_database** (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_status_db_error_on**(*mocker*, *setup_database*)

Test: Handle database error during the update process.

Steps:

1. Add a machine to the database.
2. Mock the database query to raise an SQLAlchemyError.
3. Call the *updateMachineStatusOn* method.
4. Verify the response indicates a database error.

Expected Outcome:

- The response should indicate a database error occurred.

Parameters

- **mocker** (*pytest fixture*) – Used for mocking objects and methods.
- **setup_database** (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_status_general_exception**(*mocker*, *setup_database*)

Test: Handle a general exception during the update process.

Steps:

1. Add a machine to the database.
2. Mock the database query to raise a general exception.
3. Call the *updateMachineStatusOn* method.
4. Verify the response indicates a general error.

Expected Outcome:

- The response should indicate a general error occurred.

Parameters

- **mocker** (*pytest fixture*) – Used for mocking objects and methods.

- **setup_database** (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_status_general_exception_on**(*mocker*, *setup_database*)

Test: Handle a general exception during the update process.

Steps:

1. Add a machine to the database.
2. Mock the database query to raise a general exception.
3. Call the *updateMachineStatusOn* method.
4. Verify the response indicates a general error.

Expected Outcome:

- The response should indicate a general error occurred.

Parameters

- **mocker** (*pytest fixture*) – Used for mocking objects and methods.
- **setup_database** (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_status_not_found**(*setup_database*)

Test: Attempt to update the status of a non-existent machine.

Steps:

1. Ensure the database is empty or does not contain the target machine.
2. Call the *updateMachineStatusOn* method with a non-existent machine name.
3. Verify the response indicates an error.

Expected Outcome:

- The response should indicate the machine was not found.

Parameters

setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_status_not_found_on**(*setup_database*)

Test: Attempt to update the status of a non-existent machine.

Steps:

1. Ensure the database is empty or does not contain the target machine.
2. Call the *updateMachineStatusOn* method with a non-existent machine name.
3. Verify the response indicates an error.

Expected Outcome:

- The response should indicate the machine was not found.

Parameters

setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_status_success**(*setup_database*)

Test: Successfully update the status of a machine to 'on'.

Steps:

1. Add a machine to the database with 'on' set to False.
2. Call the *updateMachineStatusOn* method with the machine name.
3. Verify the response is successful.
4. Confirm the machine's 'on' status is updated to True.

Expected Outcome:

- The machine's 'on' status is updated to True.

Parameters

setup_database (*fixture*) – The database session used for test setup.

test.test_machine.**test_update_machine_status_success_on**(*setup_database*)

Test: Successfully update the status of a machine to 'on'.

Steps:

1. Add a machine to the database with 'on' set to False.
2. Call the *updateMachineStatusOn* method with the machine name.
3. Verify the response is successful.
4. Confirm the machine's 'on' status is updated to True.

Expected Outcome:

- The machine's 'on' status is updated to True.

Parameters

setup_database (*fixture*) – The database session used for test setup.

4.6 test_residetnt module

test.test_residetnt.**test_create_resident_duplicate**(*setup_database*)

Test the behavior when attempting to create a duplicate resident in the same room.

This test verifies that the *create_resident* method of the *ResidentController* returns an error response when trying to create a resident with the same name, surname, and birth date as an existing resident in the same room. It also ensures that no duplicate resident is added to the database.

4.6.1 Parameters:

setup_database

[Session] A pytest fixture that sets up an in-memory SQLite database session for testing purposes.

4.6.2 Returns:

None

The test asserts various conditions to ensure correctness and does not return any value.

test.test_residetnt.**test_create_resident_family_no_room**(*setup_database*)

Test the behavior when trying to create a resident whose family does not have an assigned room.

This test verifies that the *create_resident* method of the *ResidentController* returns an error response if the family associated with the resident does not have an assigned room. It also ensures that no new resident is added to the database in this scenario.

4.6.3 Parameters:

setup_database
[Session] A pytest fixture that sets up an in-memory SQLite database session for testing purposes.

4.6.4 Returns:

None
The test asserts various conditions to ensure correctness and does not return any value.

test.test_residetnt.**test_create_resident_family_not_exist**(*setup_database*)

Test the behavior when trying to create a resident with a nonexistent family.

This test verifies that the *create_resident* method of the *ResidentController* returns an error response if the specified family ID does not exist in the database. It also ensures that no new resident is added to the database in this scenario.

4.6.5 Parameters:

setup_database
[Session] A pytest fixture that sets up an in-memory SQLite database session for testing purposes.

4.6.6 Returns:

None
The test asserts various conditions to ensure correctness and does not return any value.

test.test_residetnt.**test_create_resident_shelter_full**(*setup_database*)

Test the behavior when attempting to create a resident in a shelter that has reached its maximum capacity.

This test verifies that the *create_resident* method of the *ResidentController* returns an error response when the shelter's *maxPeople* limit has been reached. It ensures no additional residents are added to the database when the shelter is full.

4.6.7 Parameters:

setup_database
[Session] A pytest fixture that sets up an in-memory SQLite database session for testing purposes.

4.6.8 Returns:

None
The test asserts various conditions to ensure correctness and does not return any value.

test.test_residetnt.**test_create_resident_success**(*setup_database*)

Test the successful creation of a resident in the database.

This test verifies that a resident can be successfully created in the database given the correct setup and input data. It ensures that all necessary entities (shelter, room, family) are added and linked correctly, and that the *create_resident* method of the *ResidentController* functions as expected.

4.6.9 Parameters:

setup_database
[Session] A pytest fixture that sets up an in-memory SQLite database session for testing purposes.

4.6.10 Returns:

None
The test asserts various conditions to ensure correctness and does not return any value.

test.test_residetnt.**test_delete_resident_empty_database**(*setup_database*)

Test: Verify that the method behaves correctly when the database is empty.

Steps:

1. Ensure the database contains no residents.
2. Call the *delete_resident* method with any ID.
3. Verify the response indicates the resident was not found.

Expected Outcome:

- The method returns a "not found" status.

test.test_residetnt.**test_delete_resident_not_found**(*setup_database*)

Test: Verify that the method returns a "not found" status when the resident does not exist.

Steps:

1. Ensure the database does not contain a resident with the given ID.
2. Call the *delete_resident* method with a non-existent ID.
3. Verify the response indicates the resident was not found.

Expected Outcome:

- The method returns a "not found" status.

test.test_residetnt.**test_delete_resident_success**(*setup_database*)

Test: Verify that the method successfully deletes a resident.

Steps:

1. Add a resident to the database.
2. Call the *delete_resident* method with the resident's ID.
3. Verify the resident is removed from the database.

Expected Outcome:

- The method returns a success status.
- The resident no longer exists in the database.

test.test_residetnt.**test_delete_resident_unexpected_error**(*setup_database*, *mocker*)

Test: Verify that the method handles unexpected errors gracefully.

Steps:

1. Mock the database delete operation to raise a generic exception.
2. Call the *delete_resident* method.
3. Verify the response indicates an error occurred.

Expected Outcome:

- The method raises an exception or returns an appropriate error message.

test.test_residetnt.**test_get_resident_by_id_empty_database**(*setup_database*)

Test: Verify that the method behaves correctly when the database is empty.

Steps:

1. Ensure the database contains no residents.
2. Call the *getResidentById* method with any ID.
3. Verify the response indicates the resident is not found.

Expected Outcome:

- The method returns an error message stating “Residente no encontrado”.

test.test_residetnt.**test_get_resident_by_id_not_found**(*setup_database*)

Test: Verify that the method returns an error when the resident is not found.

Steps:

1. Ensure the database does not contain the specified resident ID.
2. Call the *getResidentById* method with a non-existent ID.
3. Verify the response indicates the resident is not found.

Expected Outcome:

- The method returns an error message stating “Residente no encontrado”.

test.test_residetnt.**test_get_resident_by_id_success**(*setup_database*)

Test: Verify that the method retrieves a resident successfully by their ID.

Steps:

1. Add a resident to the database.
2. Call the *getResidentById* method with the resident’s ID.
3. Verify the resident’s details are returned correctly.

Expected Outcome:

- The resident’s details are returned as a dictionary.

test.test_residetnt.**test_get_resident_by_id_unexpected_error**(*setup_database*, *mocker*)

Test: Verify that the method handles unexpected errors gracefully.

Steps:

1. Mock the database query to raise a generic exception.
2. Call the *getResidentById* method.

3. Verify the response indicates an error.

Expected Outcome:

- The method returns an error message indicating an unexpected error occurred.

test.test_residetnt.**test_get_resident_room_no_room_assigned**(*setup_database*)

Test: Verify that the method handles the case where the resident has no assigned room.

Steps:

1. Add a resident without an assigned room to the database.
2. Call the *getResidentRoomByNameAndSurname* method with the resident's name and surname.
3. Verify the response indicates the resident has no assigned room.

Expected Outcome:

- The method returns an error status.
- The response contains an error message indicating the resident has no assigned room.

test.test_residetnt.**test_get_resident_room_resident_not_found**(*setup_database*)

Test: Verify that the method handles the case where the resident does not exist.

Steps:

1. Ensure no resident with the specified name and surname exists in the database.
2. Call the *getResidentRoomByNameAndSurname* method with non-existent name and surname.
3. Verify the response indicates the resident is not found.

Expected Outcome:

- The method returns an error status.
- The response contains an error message indicating the resident was not found.

test.test_residetnt.**test_get_resident_room_room_not_found**(*setup_database*)

Test: Verify that the method handles the case where the resident's assigned room does not exist.

Steps:

1. Add a resident with an assigned room ID but without the corresponding room in the database.
2. Call the *getResidentRoomByNameAndSurname* method with the resident's name and surname.
3. Verify the response indicates the room was not found.

Expected Outcome:

- The method returns an error status.
- The response contains an error message indicating the room was not found.

test.test_residetnt.**test_get_resident_room_success**(*setup_database*)

Test: Verify that the method retrieves the room information for a resident with an assigned room.

Steps:

1. Add a resident and their assigned room to the database.
2. Call the *getResidentRoomByNameAndSurname* method with the resident's name and surname.
3. Verify the response contains the correct room information.

Expected Outcome:

- The method returns a success status.
- The response contains the correct room details.

test.test_residetnt.**test_list_residents**(*setup_database*)

Test the *list_residents* method to ensure it returns all residents in the database.

This test checks that the method retrieves the correct list of all residents stored in the database.

4.6.11 Steps:

1. Add multiple residents to the database.
2. Call the *list_residents* method to retrieve the list of residents.
3. Verify that the response contains all added residents with the correct details.

4.6.12 Parameters:

setup_database
[Session] A pytest fixture providing an in-memory SQLite database session for testing.

4.6.13 Returns:

None
The test asserts the expected behavior and does not return any value.

test.test_residetnt.**test_list_residents_in_room**(*setup_database*)

Test the *list_residents_in_room* method to ensure it retrieves all residents in a specified room.

This test verifies that the *list_residents_in_room* method of the *ResidentController* correctly returns a list of residents assigned to a given room, and that residents in other rooms are not included in the response.

4.6.14 Steps:

1. Add multiple rooms and residents to the database.
2. Call the *list_residents_in_room* method with the ID of a specific room.
3. Verify that the returned data includes only the residents assigned to that room.

4.6.15 Parameters:

setup_database
[Session] A pytest fixture providing an in-memory SQLite database session for testing.

4.6.16 Returns:

None
The test asserts various conditions to ensure correctness and does not return any value.

test.test_residetnt.**test_list_residents_in_room_database_error**(*setup_database*, *mocker*)

Test: Verify that the method handles database errors gracefully.

Steps:

1. Mock the database session to raise an exception during the query.
2. Call the *list_residents_in_room* method.

3. Verify the response indicates a database error.

Expected Outcome:

- The method returns an error status.
- The response contains an error message.

test.test_residetnt.**test_list_residents_in_room_no_residents**(*setup_database*)

Test: Verify that the method returns an empty list when no residents are in the specified room.

Steps:

1. Add a room without residents to the database.
2. Call the *list_residents_in_room* method for the specific room.
3. Verify the response contains an empty list.

Expected Outcome:

- The method returns a success status.
- The response contains an empty list.

test.test_residetnt.**test_list_residents_in_room_room_not_found**(*setup_database*)

Test: Verify that the method handles the case where the specified room does not exist.

Steps:

1. Ensure the database does not contain the specified room.
2. Call the *list_residents_in_room* method with a non-existent room ID.
3. Verify the response contains an empty list.

Expected Outcome:

- The method returns a success status.
- The response contains an empty list.

test.test_residetnt.**test_list_residents_in_room_success**(*setup_database*)

Test: Verify that the method successfully retrieves all residents in a specific room.

Steps:

1. Add a room and multiple residents to the database.
2. Call the *list_residents_in_room* method for the specific room.
3. Verify the response includes the correct residents.

Expected Outcome:

- The method returns a success status.
- The response contains a list of residents assigned to the specified room.

test.test_residetnt.**test_list_residents_no_residents**(*setup_database*)

Test the *list_residents* method to ensure it returns an empty list when no residents are present in the database.

This test verifies the behavior of the method when the database contains no resident records.

4.6.17 Steps:

1. Ensure the database is empty with no residents added.
2. Call the *list_residents* method to retrieve the list of residents.
3. Verify that the response contains an empty list.

4.6.18 Parameters:

setup_database
[Session] A pytest fixture providing an in-memory SQLite database session for testing.

4.6.19 Returns:

None
The test asserts the expected behavior and does not return any value.

`test.test_residetnt.`**`test_login_empty_database`**(*setup_database*)

Test: Verify that the method behaves correctly when the database is empty.

Steps:

1. Ensure the database contains no residents.
2. Call the *login* method with any name and surname.
3. Verify the response indicates invalid credentials.

Expected Outcome:

- The method returns an error message stating “Invalid credentials”.

`test.test_residetnt.`**`test_login_invalid_credentials`**(*setup_database*)

Test: Verify that the method returns an error when the credentials are invalid.

Steps:

1. Ensure the database does not contain the provided name and surname.
2. Call the *login* method with invalid credentials.
3. Verify the response indicates invalid credentials.

Expected Outcome:

- The method returns an error message stating “Invalid credentials”.

`test.test_residetnt.`**`test_login_success`**(*setup_database*)

Test: Verify that the method successfully logs in a resident with valid credentials.

Steps:

1. Add a resident to the database.
2. Call the *login* method with the resident’s name and surname.
3. Verify the response includes the resident’s ID, name, and surname.

Expected Outcome:

- The login is successful, and the resident’s details are returned.

test.test_residetnt.**test_login_unexpected_error**(*setup_database*, *mocker*)

Test: Verify that the method handles unexpected errors gracefully.

Steps:

1. Mock the database query to raise a generic exception.
2. Call the *login* method.
3. Verify the response indicates an error.

Expected Outcome:

- The method returns an error message indicating an unexpected error occurred.

test.test_residetnt.**test_update_resident_birthdate_database_error**(*setup_database*, *mocker*)

Test: Verify that the method handles database errors gracefully.

Steps:

1. Mock the database session to raise an SQLAlchemyError.
2. Call the *updateResidentBirthDate* method.
3. Verify the response indicates a database error.

Expected Outcome:

- The method returns an error message indicating a database issue.

test.test_residetnt.**test_update_resident_birthdate_invalid_format**(*setup_database*)

Test: Verify that the method handles invalid date formats gracefully.

Steps:

1. Add a resident to the database.
2. Call the *updateResidentBirthDate* method with an invalid date format.
3. Verify the response indicates a conversion error.

Expected Outcome:

- The method returns an error message indicating the date format is invalid.

test.test_residetnt.**test_update_resident_birthdate_not_found**(*setup_database*)

Test: Verify that the method handles the case when a resident is not found.

Steps:

1. Ensure the database does not contain the specified resident.
2. Call the *updateResidentBirthDate* method with a non-existent resident ID.
3. Verify the response indicates the resident is not found.

Expected Outcome:

- The method returns an error message indicating the resident is not found.

test.test_residetnt.**test_update_resident_birthdate_success**(*setup_database*)

Test: Verify that the method successfully updates a resident's birth date.

Steps:

1. Add a resident to the database.
2. Call the *updateResidentBirthDate* method with a valid new birth date.

3. Verify the resident's birth date is updated in the database.

Expected Outcome:

- The method returns a success message.
- The resident's birth date is updated in the database.

test.test_residetnt.**test_update_resident_gender_database_error**(*setup_database*, *mocker*)

Test: Verify that the method handles database errors gracefully.

Steps:

1. Mock the database session to raise an SQLAlchemyError.
2. Call the *updateResidentGender* method.
3. Verify the response indicates a database error.

Expected Outcome:

- The method returns an error message indicating a database issue.

test.test_residetnt.**test_update_resident_gender_invalid_gender**(*setup_database*)

Test: Verify that the method handles invalid gender values gracefully.

Steps:

1. Add a resident to the database.
2. Call the *updateResidentGender* method with an invalid gender value.
3. Verify the response indicates the gender is invalid.

Expected Outcome:

- The method returns an error message indicating the gender is invalid.

test.test_residetnt.**test_update_resident_gender_not_found**(*setup_database*)

Test: Verify that the method handles the case when a resident is not found.

Steps:

1. Ensure the database does not contain the specified resident.
2. Call the *updateResidentGender* method with a non-existent resident ID.
3. Verify the response indicates the resident is not found.

Expected Outcome:

- The method returns an error message indicating the resident is not found.

test.test_residetnt.**test_update_resident_gender_success**(*setup_database*)

Test: Verify that the method successfully updates a resident's gender.

Steps:

1. Add a resident to the database.
2. Call the *updateResidentGender* method with a valid new gender.
3. Verify the resident's gender is updated in the database.

Expected Outcome:

- The method returns a success message.
- The resident's gender is updated in the database.

test.test_residetnt.**test_update_resident_room_database_error**(*setup_database*, *mocker*)

Test: Verify that the method handles database errors gracefully.

Steps:

1. Mock the database session to raise an SQLAlchemyError.
2. Call the *updateResidentRoom* method.
3. Verify the response indicates a database error.

Expected Outcome:

- The method returns an error status.
- The response contains an error message indicating a database issue.

test.test_residetnt.**test_update_resident_room_resident_not_found**(*setup_database*)

Test: Verify that the method handles the case where the resident does not exist.

Steps:

1. Ensure no resident with the specified ID exists in the database.
2. Call the *updateResidentRoom* method with a non-existent resident ID.
3. Verify the response indicates the resident is not found.

Expected Outcome:

- The method returns an error status.
- The response contains an error message indicating the resident was not found.

test.test_residetnt.**test_update_resident_room_room_not_found**(*setup_database*)

Test: Verify that the method updates the resident's room even if the new room does not exist in the database.

Steps:

1. Add a resident and a single room to the database.
2. Call the *updateResidentRoom* method with a non-existent room ID.
3. Verify the resident's *idRoom* field is updated to the new room ID.

Expected Outcome:

- The method returns a success status.
- The resident's *idRoom* field is updated to the specified new room ID, even if the room does not exist.

test.test_residetnt.**test_update_resident_room_success**(*setup_database*)

Test: Verify that the method successfully updates the resident's room.

Steps:

1. Add a resident and two rooms to the database.
2. Call the *updateResidentRoom* method to assign the resident to a new room.
3. Verify the room ID is updated correctly.

Expected Outcome:

- The method returns a success status.
- The resident's *idRoom* field is updated to the new room ID.

test.test_residetnt.**test_update_resident_surname_database_error**(*setup_database*, *mocker*)

Test: Verify that the method handles database errors gracefully.

Steps:

1. Mock the database session to raise an SQLAlchemyError.
2. Call the *updateResidentSurname* method.
3. Verify the response indicates a database error.

Expected Outcome:

- The method returns an error message indicating a database issue.

test.test_residetnt.**test_update_resident_surname_not_found**(*setup_database*)

Test: Verify that the method returns an error when the resident is not found.

Steps:

1. Ensure the database does not contain the specified resident.
2. Call the *updateResidentSurname* method with a non-existent ID.
3. Verify the response indicates the resident is not found.

Expected Outcome:

- The method returns an error message stating "Residente no encontrado".

test.test_residetnt.**test_update_resident_surname_success**(*setup_database*)

Test: Verify that the method successfully updates a resident's surname.

Steps:

1. Add a resident to the database.
2. Call the *updateResidentSurname* method with a new surname.
3. Verify the resident's surname is updated.

Expected Outcome:

- The resident's surname is updated successfully in the database.

4.7 test_room module

test.test_room.**test_access_room_family_room**(*setup_database*)

Test: Verify that access is granted when a resident enters their assigned family room.

Steps:

1. Add a family with an assigned room to the database.
2. Add a resident belonging to that family.
3. Add the corresponding room with sufficient capacity.
4. Call the *access_room* method with the resident's ID and the room ID.
5. Verify that access is granted with the appropriate success message.

Expected Outcome:

- The method should return "Access granted. Welcome to the room." when the resident accesses their assigned family room.

test.test_room.**test_access_room_full_room**(*setup_database*)

Test: Verify that access is denied when the room is at full capacity.

Steps:

1. Add a room with a maximum capacity of 2 to the database.
2. Add two residents assigned to the room to fill it to capacity.
3. Add a new resident who will attempt to access the full room.
4. Call the *access_room* method with the new resident's ID and the room ID.
5. Verify that the method denies access with the appropriate error message.

Expected Outcome:

- The method should return "Access denied. La sala está llena." when a resident tries to enter a room that has reached its maximum capacity.

test.test_room.**test_access_room_maintenance_room**(*setup_database*)

Test: Verify that access to a maintenance room is denied.

Steps:

1. Add a valid resident to the database.
2. Add a room named "Mantenimiento" to the database.
3. Call the *access_room* method with the resident ID and the maintenance room ID.
4. Verify that the method denies access with the appropriate error message.

Expected Outcome:

- The method should return "Access denied. No puedes entrar a la sala de mantenimiento." when a resident tries to access a maintenance room.

test.test_room.**test_access_room_public_room**(*setup_database*)

Test: Verify that access is granted when the resident attempts to enter a public room.

Steps:

1. Add a resident to the database.
2. Add a room labeled as a "Common Area" with sufficient capacity.
3. Call the *access_room* method with the resident's ID and the room ID.
4. Verify that access is granted with the appropriate success message.

Expected Outcome:

- The method should return "Access granted. Welcome to the room." when a resident successfully accesses a public room.

test.test_room.**test_access_room_resident_not_found**(*setup_database*)

Test: Verify that attempting to access a room with a non-existent resident returns the correct error message.

Steps:

1. Use a resident ID that does not exist in the database.
2. Call the *access_room* method with the invalid resident ID and a valid room ID.
3. Verify that the method returns the appropriate error message.

Expected Outcome:

- The method should return “Resident not found.” when the resident ID does not exist.

test.test_room.**test_access_room_room_not_found**(*setup_database*)

Test: Verify that attempting to access a non-existent room returns the correct error message.

Steps:

1. Add a valid resident to the database.
2. Use a room ID that does not exist in the database.
3. Call the *access_room* method with the valid resident ID and invalid room ID.
4. Verify that the method returns the appropriate error message.

Expected Outcome:

- The method should return “Room not found.” when the room ID does not exist.

test.test_room.**test_access_room_wrong_room**(*setup_database*)

Test: Verify that access is denied when a resident attempts to enter a room not assigned to their family.

Steps:

1. Add two families, each assigned to a different room.
2. Add a resident belonging to the first family.
3. Add a room associated with the second family.
4. Call the *access_room* method with the resident’s ID and the second room’s ID.
5. Verify that access is denied with the appropriate error message.

Expected Outcome:

- The method should return “Access denied. You are in the wrong room.” when the resident attempts to access a room they are not assigned to.

test.test_room.**test_create_room_admin_not_exist**(*setup_database*)

Test: Prevent creating a room when the admin does not exist.

Steps:

1. Call the *create_room* method with a nonexistent admin ID.
2. Verify the response indicates the admin does not exist.

Expected Outcome:

- The operation should fail with an error message about the admin.

test.test_room.**test_create_room_duplicate**(*setup_database*)

Test: Prevent creating a duplicate room in the same shelter.

Steps:

1. Add a room to the database.
2. Attempt to add another room with the same name in the same shelter.
3. Verify the response indicates duplication.

Expected Outcome:

- The operation should fail with an error message about the duplicate room.

test.test_room.**test_create_room_shelter_not_exist**(*setup_database*)

Test: Prevent creating a room when the shelter does not exist.

Steps:

1. Call the *create_room* method with a nonexistent shelter ID.
2. Verify the response indicates the shelter does not exist.

Expected Outcome:

- The operation should fail with an error message about the shelter.

test.test_room.**test_create_room_success**(*setup_database*)

Test: Successfully create a new room.

Steps:

1. Add an admin and shelter to the database.
2. Call the *create_room* method with valid data.
3. Verify the response indicates success.
4. Confirm the room exists in the database with correct details.

Expected Outcome:

- The room should be successfully created and associated with the shelter.

test.test_room.**test_list_rooms_empty**(*setup_database*)

Test: Validate behavior when no rooms are present in the database.

Steps:

1. Ensure the database is empty.
2. Call the *list_rooms* method.

Expected Outcome:

- The method returns an empty list.

test.test_room.**test_list_rooms_room_no_matches**(*setup_database*)

Test: No rooms with names starting with “Room”.

Steps:

1. Add rooms to the database with names not starting with “Room”.
2. Call *list_rooms_Room* to fetch the rooms.
3. Verify the response is an empty list.

Expected Outcome:

- Response is an empty list.

test.test_room.**test_list_rooms_room_success**(*setup_database*)

Test: Successfully list rooms with names starting with “Room”.

Steps:

1. Add multiple rooms to the database, including ones with and without names starting with “Room”.
2. Call *list_rooms_Room* to fetch the rooms.
3. Verify the response contains only rooms whose names start with “Room”.

Expected Outcome:

- Response contains only rooms with names starting with “Room”.

test.test_room.**test_list_rooms_room_unexpected_exception**(*mocker*, *setup_database*)

Test: Simulate an unexpected exception during the operation.

Steps:

1. Mock the session to raise a generic exception when querying the database.
2. Call *list_rooms_Room*.
3. Verify the response indicates an error occurred.

Expected Outcome:

- Response indicates an error occurred.

test.test_room.**test_list_rooms_sqlalchemy_error**(*mocker*, *setup_database*)

Test: Validate behavior when a SQLAlchemyError occurs during the query.

Steps:

1. Mock the session to raise a SQLAlchemyError during query execution.
2. Call the *list_rooms* method.

Expected Outcome:

- The method returns a status of “error” with the error message.

test.test_room.**test_list_rooms_success**(*setup_database*)

Test: Validate successful retrieval of rooms from the database.

Steps:

1. Add multiple rooms to the database.
2. Call the *list_rooms* method.

Expected Outcome:

- The method returns a list of rooms with correct details.

test.test_room.**test_list_rooms_unexpected_exception**(*mocker*, *setup_database*)

Test: Validate behavior when an unexpected exception occurs.

Steps:

1. Mock the session to raise a generic exception during query execution.
2. Call the *list_rooms* method.

Expected Outcome:

- The method returns a status of “error” with the exception message.

test.test_room.**test_list_rooms_with_resident_count**(*setup_database*)

Test: List all rooms along with the count of residents in each room.

Steps:

1. Add rooms and residents to the database.
2. Call the *list_rooms_with_resident_count* method.
3. Verify the returned data matches the database records.

Expected Outcome:

- The room list should include the correct count of residents for each room.

test.test_room.**test_update_room_name_room_not_found**(*setup_database*)

Test: Attempt to update the name of a non-existent room.

Steps:

1. Ensure the database is empty.
2. Call *updateRoomName* with a non-existent room ID.
3. Verify the response indicates the room was not found.

Expected Outcome:

- Response indicates the room was not found.

test.test_room.**test_update_room_name_sqlalchemy_error**(*mocker*, *setup_database*)

Test: Simulate a SQLAlchemyError during the operation.

Steps:

1. Mock the session to raise a SQLAlchemyError when querying the database.
2. Call *updateRoomName*.
3. Verify the response indicates a database error.

Expected Outcome:

- Response indicates a database error occurred.

test.test_room.**test_update_room_name_success**(*setup_database*)

Test: Successfully update the name of an existing room.

Steps:

1. Add a room to the database with an initial name.
2. Call *updateRoomName* to update the room's name.
3. Verify the response and ensure the name is updated in the database.

Expected Outcome:

- Response indicates success.
- The room's name is updated in the database.

test.test_room.**test_update_room_name_unexpected_exception**(*mocker*, *setup_database*)

Test: Simulate an unexpected exception during the operation.

Steps:

1. Mock the session to raise a generic exception when querying the database.
2. Call *updateRoomName*.
3. Verify the response indicates an unexpected error occurred.

Expected Outcome:

- Response indicates an unexpected error occurred.

4.8 test_shelter module

test.test_shelter.**test_get_shelter_energy_level_success**(*setup_database*)

Test: Retrieve the energy level of a shelter.

Steps:

1. Add a shelter with specific energy level to the database.
2. Call the *get_shelter_energy_level* method.
3. Verify the returned energy level matches the database record.

Expected Outcome:

- The energy level should be correctly retrieved.

test.test_shelter.**test_get_shelter_no_shelter_found**(*setup_database*)

Test: Handle the case where no shelter is found in the database.

Steps:

1. Ensure the shelter table is empty.
2. Call any of the methods (*get_shelter_energy_level*, *get_shelter_water_level*, or *get_shelter_radiation_level*).
3. Verify that the method raises a *ValueError*.

Expected Outcome:

- A *ValueError* should be raised with the appropriate message.

test.test_shelter.**test_get_shelter_radiation_level_success**(*setup_database*)

Test: Retrieve the radiation level of a shelter.

Steps:

1. Add a shelter with specific radiation level to the database.
2. Call the *get_shelter_radiation_level* method.
3. Verify the returned radiation level matches the database record.

Expected Outcome:

- The radiation level should be correctly retrieved.

test.test_shelter.**test_get_shelter_water_level_success**(*setup_database*)

Test: Retrieve the water level of a shelter.

Steps:

1. Add a shelter with specific water level to the database.
2. Call the *get_shelter_water_level* method.
3. Verify the returned water level matches the database record.

Expected Outcome:

- The water level should be correctly retrieved.

test.test_shelter.**test_updateShelterEnergyLevel_success**(*setup_database*, *mocker*)

Test to verify that the updateShelterEnergyLevel method correctly updates the energy level of a shelter.

This test simulates the process of updating the energy level of a shelter in the database. It first creates a shelter with an initial energy level of 100, saves it to the database, and then calls the updateShelterEnergyLevel method to update the energy level to 80.

Steps:

1. Add a shelter to the database.
2. Call the *updateShelterEnergyLevel* method to update the shelter's energy level.
3. Verify the response and the updated energy level in the database.

Expected Outcome:

- The method returns a success message.
- The shelter's energy level is updated in the database.

test.test_shelter.**test_updateShelterRadiationLevel_success**(*setup_database*)

Test to verify that the updateShelterRadiationLevel method correctly updates the radiation level of a shelter.

This test simulates the process of updating the radiation level of a shelter in the database. It first creates a shelter with an initial radiation level of 10, saves it to the database, and then calls the updateShelterRadiationLevel method to update the radiation level to 8.

Steps:

1. Add a shelter to the database.
2. Call the *updateShelterRadiationLevel* method to update the shelter's radiation level.
3. Verify the response and the updated radiation level in the database.

Expected Outcome:

- The method returns a success message.
- The shelter's radiation level is updated in the database.

test.test_shelter.**test_updateShelterWaterLevel_success**(*setup_database*, *mocker*)

Test to verify that the updateShelterWaterLevel method correctly updates the water level of a shelter.

This test simulates the process of updating the water level of a shelter in the database. It first creates a shelter with an initial water level of 100, saves it to the database, and then calls the updateShelterWaterLevel method to update the water level to 80.

Steps:

1. Add a shelter to the database.
2. Call the *updateWaterEnergyLevel* method to update the shelter's energy level.
3. Verify the response and the updated water level in the database.

Expected Outcome:

- The method returns a success message.
- The shelter's water level is updated in the database.

CHAPTER

FIVE

INDICES

- modindex

PYTHON MODULE INDEX

a

t