Sync Smart Home API Docs

None

None

None

Table of contents

Sync Smart Home API Documentation	3
2. API Reference	4
2.1 backend.main	4
2.2 backend.routes.user_routes	5
2.3 backend.routes.data_routes	7
2.4 backend.routes.report_routes	11
2.5 backend.models.device	13
2.6 backend.models.energy_data	14
2.7 backend.models.energy_summary	15
2.8 backend.models.profile	16
2.9 backend.models.room	17
2.10 backend.models.schedule	18
2.11 backend.models.user	18
2.12 backend.db.database	22
2.13 backend.core.config	23
2.14 backend.core.security	23

1. Sync Smart Home API Documentation

2. API Reference

2.1 backend.main

This module initializes & configures the FastAPI application

It acts as a middleware between the front-end & database, handling: - API route inclusion - CORS middleware for front-end communication - Database initialization & application life-cycle management

2.1.1 backend.main.RootResponse

Bases: BaseModel

Response model for the root endpoint

Attributes:

Name	Туре	Description
message	str	Welcome message

2.1.2 backend.main.lifespan(app context) async

Defines application's lifespan event handler

- Initialization the database at startup
- \bullet Sets up application-wide state variables
- Logs startup & shutdown events

Parameters:

Name	Туре	Description	Default
app_context	FastAPI	The FastAPI application instance	required

Yields:

Name	Туре	Description
None		Allows FastAPI to manage application life-cycle

Туре	Description
Exception	If database initialization fails

```
Source code in backend/main.py
                                          @asynccontextmanager
async def lifespan(app_context: FastAPI):
    """
40
41
42
43
                                                                            Defines application's lifespan event handler
                                                                          - Initialization the database at startup
  44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
                                                                          - Sets up application-wide state variables
- Logs startup & shutdown events
                                                                                                              app_context (FastAPI): The FastAPI application instance
                                                                                                        None: Allows FastAPI to manage application life-cycle
                                                                            Exception: If database initialization fails """ % \left( \frac{1}{2}\right) =\frac{1}{2}\left( \frac{1}{2}\right) \left( \frac{1}{2}
                                                                        # Placeholder for future app-wide state
                                                                                            ccept Exception as e:
    logger.error("Failed to initialize the database: %e.")
  63
64
  65
66
                                                                                  logger.info("Application is shutting down.")
```

2.1.3 backend.main.read_root()

Root endpoint

Returns:

Name	Туре	Description
RootResponse	RootResponse	A welcome message for the API

2.2 backend.routes.user routes

This module defines user-related API routes for registration, authentication & role-based access

It includes: - User registration with hashed password storage - User authentication & JWT token issuance - Admin dashboard access (restricted to users with the "admin" role)

$2.2.1 \verb| backend.routes.user_routes.get_admin_dashboard() | \verb| async| \\$

Admin dashboard endpoint (restricted access)

Returns:

Name	Туре	Description
dict		A welcome message confirming admin access

2.2.2 backend.routes.user_routes.login(form_data=Depends()) async

Authenticate user & return a JWT token

Parameters:

Name	Туре	Description	Default
form_data	OAuth2PasswordRequestForm	User-provided login credentials	Depends()

Returns:

Nam	е Туре	Description	
dict		A dictionary containing the access token & token type	

Raises:

Туре	Description
HTTPException(400)	If the username or password is incorrect

```
Source code in backend/routes/user_routes.py

67
68 async def login(form_data: OAuth2PasswordRequestForm = Depends()):
69
70    Authenticate user & return a JWT token
71
72    Args:
73    form_data (OAuth2PasswordRequestForm): User-provided login credentials
74
75    Returns:
76    dict: A dictionary containing the access token & token type
77
78    Raises:
79    HTTPException (400): If the username or password is incorrect
80
81
82
83
84
85    access_token = create_access_token(
86    data = ""sub": user("email"; form_data.username))
85    access_token = create_access_token(
86    data = ""sub": user("email"), "role": user.get("role", "user"),
87    expires_delta = timedelta(minutes = 30)
88
89
90    return ("access_token": access_token, "token_type": "bearer")
```

 $2.2.3 \ \texttt{backend.routes.user_routes.register_user(user, \ \texttt{background_tasks}) \ \texttt{async}$

Register a new user in the database

Name	Туре	Description	Default
user	UserCreate	User registration data including email, password $\&$ optional role	required

Returns:

Name	Туре	Description
UserResponse		The created user information excluding password

Raises:

Туре	Description	
HTTPException(400)	If the email is already registered	
HTTPException(500)	If there is an error inserted the user into the database	

```
Source code in backend/routes/user_routes.py
@router.post("/register", response_model = UserResponse)
          async def register_user(user: UserCreate, background_tasks: BackgroundTasks):
                Register a new user in the database
                      UserResponse: The created user information excluding password
                HTTPException (400): If the email is already registered
HTTPException (500): If there is an error inserted the user into the database
               # Check if email exists already
if users_collection.find_one(("email": user.email)):
    raise HTTPException(status_code = 400, detail = "Registration failed, please try again.")
                # Prepare user data for insertion
                user_data = {
   "email": user.email,
                     "password_hash": hash_password(user.password),
                     "is_verified": False,
"created_at": datetime.now(timezone.utc),
"updated_at": datetime.now(timezone.utc),
"role": user.role or "user"
                # Database user insert
                     result = users_collection.insert_one(user_data)
                except Exception as exc:
    raise HTTPException(status_code = 500, detail = f"Failed to register user: {exc}") from exc
               # Generate verification token & schedule when verification email sends token = generate_verification_token(user.email) background_tasks.add_task(send_verification_email, user.email, token)
                 id = str(result.inserted_id),
role = user_data["role"],
email = user.email,
is_verified = False,
                     created_at = user_data["created_at"]
```

2.3 backend.routes.data_routes

This module defines API routes for managing & retrieving energy consumption data

It includes: - An endpoint for adding new energy data (admin-only access) - An endpoint for fetching aggregated energy consumption data with filtering options - An admin dashboard route (restricted to users with an "admin" role)

2.3.1 backend.routes.data routes.add energy data(data) async

Add new energy consumption data to the database (admin-only)

Parameters:

Name	Туре	Description	Default
data	EnergyData	The energy data record to be added	required

Returns:

Name	Туре	Description
dict		A confirmation message upon successfully insertion

```
Source code in backend/routes/data_routes.py

grouter.post("/add", dependencies = [Depends(role_required("admin"))])

async def add_energy_data(data: EnergyData):

"""

Add new energy consumption data to the database (admin-only)

Args:

data (EnergyData): The energy data record to be added

Returns:

dict: A confirmation message upon successfully insertion

"""

energy_collection.insert_one(data.model_dump())  # Insert into MongoDB

return {"message": "Energy data added successfully"}
```

 $2.3.2 \; \texttt{backend.routes.data_routes.get_admin_dashboard()} \; \; \texttt{async}$

Admin dashboard endpoint (restricted access)

Returns:

Name	Туре	Description
dict		A welcome message confirming admin access

2.3.3 backend.routes.data_routes.get_aggregated_data(start_date=Query(None, description='Start date (YYYY-MM-DD)'), end_date=Query(None, description='End date (YYYY-MM-DD)'), device_id=Query(None, description='Device ID filter'), location=Query(None, description='Location filter'), interval='day') async

Retrieve aggregated energy consumption data based on time interval & filters

Name	Туре	Description	Default
start_date	Optional[str]	The start date for filtering (YYYY-MM-DD)	<pre>Query(None, description='Start date (YYYY-MM-DD)')</pre>
end_date	Optional[str]	The end date for filtering (YYYY-MM-DD)	<pre>Query(None, description='End date (YYYY-MM-DD)')</pre>
device_id	Optional[str]	The ID of the device to filter data	<pre>Query(None, description='Device ID filter')</pre>
location	Optional[str]	The location filter	<pre>Query(None, description='Location filter')</pre>
interval	<pre>Literal['hour', 'day', 'week']</pre>	The time interval for aggregation (defaults to "day")	'day'

Returns:

Name	Туре	Description
dict		Aggregated energy consumption data grouped by the selected interval

```
Source code in backend/routes/data routes.py
         @router.get("/aggregate")
        grouter.get("/aggregated]
async def get_aggregated_data(
    start_date: Optional[str] = Query(None, description = "Start_date (YYYY-MM-DD)"),
    end_date: Optional[str] = Query(None, description = "End_date (YYYY-MM-DD)"),
    device_id: Optional[str] = Query(None, description = "Device ID filter"),
    location: Optional[str] = Query(None, description = "Location filter"),
              interval: Literal["hour", "day", "week"] = "day",
              Retrieve aggregated energy consumption data based on time interval & filters
                    start_date (Optional[str]): The start date for filtering (YYYY-MM-DD)
end_date (Optional[str]): The end date for filtering (YYYY-MM-DD)
                   device id (Optional[str]): The ID of the device to filter data location (Optional[str]): The location filter
                    interval (Literal["hour", "day", "week"]): The time interval for aggregation (defaults to "day")
 84
85
            dict: Aggregated energy consumption data grouped by the selected interval
 88
89
                query["timestamp"]
                       "$gte": datetime.strptime(start_date, "%Y-%m-%d"),
"$lte": datetime.strptime(end_date, "%Y-%m-%d")
            if device_id:
           query["device_id"] = device_id
if location:
                    query["location"] = location
           time_group = {
   "year": {"$year": "$timestamp"},
                 "month": {"$month": "$timestamp"}
                   "day": {"$dayOfMonth": "$timestamp"}
105
           if interval == "hour":
107
108
                            "device_id": "$device_id",
109
110
                       "year": {"$year": "$timestamp"},
"month": {"$month": "$timestamp"},
                         "day": {"$dayOfMonth": "$timestamp"},
"hour": {"$hour": "$timestamp"}
           elif interval == "day":
115
116
                   time_group = {
   "device_id": "$device_id",
                         "year": ("$year": "$timestamp"),
"month": ("$month": "$timestamp"),
"day": ("$dayOfMonth": "$timestamp")
           elif interval == "week":
            time_group = {
    "device_id": "$device_id",
    "year": ("$year": "$timestamp"),
    "week": ("$isoWeek": "$timestamp")
128
129
             aggregation_pipeline = [
             "device_id": "$device_id",  # Group by device

**time_group  # Group by time interval
                               **time_group
                          }, "total_energy": {"$sum": "$energy_consumed"}
138
             result = list(energy_collection.aggregate(aggregation_pipeline))
140
              return {"aggregated_data": result}
```

2.3.4 backend.routes.data_routes.get_user_energy_data(start_date=Query(None, description='Start date (YYYY-MM-DD)'), end_date=Query(None, description='End date (YYYY-MM-DD)'), device_id=Query(None, description='Device ID filter'), =Depends(profile permission required('can access energy data'))) async

Retrieve energy data for profiles with energy data access permission

Name	Туре	Description	Default
start_date	Optional[str]	Start date for filtering	Query(None, description='Start date (YYYY-MM-DD)')
end_date	Optional[str]	End date for filtering	Query(None, description='End date (YYYY-MM-DD)')
device_id	Optional[str]	Device ID filter	Query(None, description='Device ID filter')

Returns:

Name	Туре	Description
dict		Energy consumption data

```
Source code in backend/routes/data_routes.py
        async def get_user_energy_data(

async def get_user_energy_data(

async def get_user_energy_data(

start_date: Optional[str] = Query(None, description = "Start_date (YYYY-MM-DD)"),

end_date: Optional[str] = Query(None, description = "End_date (YYYY-MM-DD)"),

device_id: Optional[str] = Query(None, description = "Device ID filter"),

= Depends(profile_permission_required("can_access_energy_data"))

...
21
22
23
24
25
26
27
28
                Retrieve energy data for profiles with energy data access permission
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
                    start_date (Optional[str]): Start date for filtering
                      end_date (Optional[str]): End date for filtering
device_id (Optional[str]): Device ID filter
               Returns:
               dict: Energy consumption data
               # Similar to existing get_aggregated_data but with profile permission check
               query = {}
             if start_date and end_date:
               query["timestamp"] = {
    "$gte": datetime.strptime(start_date, "%Y-%m-%d"),
    "$lte": datetime.strptime(end_date, "%Y-%m-%d")
                     query["device_id"] = device_id
48
49
50
51
                energy_data = list(energy_collection.find(query, {"_id": 0}))
               return {"data": energy_data}
```

2.4 backend.routes.report_routes

This module provides API endpoints for generating energy consumption reports

It includes: - An endpoint to generate reports in CSV or PDF format, optionally filtered by a data range - Reports are stored in the generated_reports_directory

```
2.4.1 backend.routes.report_routes.generate_report(format=Query('csv', enum=['csv', 'pdf']),
start_date=Query(None, description='Start date (YYYY-MM-DD)'), end_date=Query(None,
description='End date (YYYY-MM-DD)')) async
```

Generate an energy consumption report in CSV or PDF format

Name	Туре	Description	Default
format	str	The desired report format, either "csv" or "pdf"	Query('csv', enum=['csv', 'pdf'])
start_date	str	The start date for filtering data (YYYY-MM-DD)	Query(None, description='Start date (YYYY-MM-DD)')
end_date	str	The end date for filtering data (YYYY-MM-DD)	Query(None, description='End date (YYYY-MM-DD)')

Returns:

Name	Туре	Description
FileResponse		The generated report file

Туре	Description	
HTTPException(400)	If an invalid date format is provided	
HTTPException(404)	If no energy data is available for the selected range	

```
Source code in backend/routes/report routes.py
       @router.post("/report", dependencies = [Depends(role_required("admin"))])
       async def generate report(
    format: str = Query("csv", enum = ["csv", "pdf"]),
    start_date: str = Query(None, description = "Start date (YYYY-MM-DD)"),
                   end_date: str = Query(None, description = "End date (YYYY-MM-DD)")
                   format (str): The desired report format, either "csv" or "pdf" start_date (str, optional): The start date for filtering data (YYYY-MM-DD)
                   end_date (str, optional): The end date for filtering data (YYYY-MM-DD)
                  HTTPException (400): If an invalid date format is provided
                   HTTPException (404): If no energy data is available for the selected range
                   energy data = get energy data(start date, end date)
                  raise HTTPException(status code = 400, detail = str(exc)) from exc
                    \texttt{raise HTTPException(status\_code = 404, detail = "No energy data available for the selected range.")} 
            timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
filename = f"{REPORTS_DIR}/energy_report_{timestamp}.{format}"
                   # Convert data to a DataFrame & format timestamps
              # df = pd.DataFrame(energy_data)
                  # df["timestamp"] = pd.to datetime(df["timestamp"]).dt.strftime("%Y-%m-%d %H:%M:%S") # Format timestamp
                                          ["Device ID", "Timestamp", "Energy Consumed (kWh)", "Location"]
                  df = pd.DataFrame(energy_data)
df = pd.DataFrame(energy_data)
df.drop(columns=["timestamps"], errors="ignore", inplace=True)  # Drop extra column
df.columns = ["Device ID", "Timestamp", "Energy Consumed (kWh)", "Location"]
                    df.to_csv(filename, index = False)
           elif format == "pdf":

# Create a PDF report with a structure table
doc = SimpleDocTemplate(filename, pagesize=letter)
                  elements = []
                  # Prepare table data
data = [["Device ID", "Timestamp", "Energy Cosumed (kWh)", "Location"]]
                   for entry in energy_data:
                        data.append([
                             str(entry.get("device_id", "N/A")),
str(entry.get("timestamp", "").strftime('%Y-%m-%d %H:%M:%S') if entry.get("timestamp") else "N/A"),
str(entry.get("energy_consumed", "N/A")),
                              str(entry.get("location", "N/A"))
80
                # Create & style table
                 table = Table(data)
table.setStyle(TableStyle([
    ('BACKGROUND', (0, 0), (-1, 0), colors.grey),
    ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),
    ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
    ('FONTNAME', (0, 0), (-1, 0), 'Belvetica-Bold'),
    ('FONTSIZE', (0, 0), (-1, 0), 12),
    ('BOTTOMPADDING', (0, 0), (-1, 0), 10),
    ('GRID', (0, 0), (-1, -1), 1, colors.black),
    ('ROWBACKGROUNDS', (0, 1), (-1, -1), [colors.whitesmoke, colors.lightgrey])
)))
                  elements.append(table)
                  doc.build(elements)
                path = os.path.abspath(filename),
filename = os.path.basename(filename),
                   media type = "application/octet-stream"
```

2.5 backend.models.device

This module defines the EnergySummary model for analyzing & reporting energy consumption

The EnergySummary model: - Aggregates energy consumption data over specified time periods - Supports reporting & historical data analysis requirements - Enables cost estimation based on energy usage - Facilitates comparisons between current & past usage periods - Provides the data foundation for energy-saving suggestions & insights

2.5.1 backend.models.device.Device

Bases: BaseModel

Pydantic model representing a smart device within the home automation system

Tracks device information, current status, & location within the home

Used for device control, automation, & energy consumption tracking

Attributes:

Name	Туре	Description
id	str	Unique identifier for the device
name	str	User-friendly name for the device
type	str	Category of device (e.g., "light", "thermostat", "appliance")
room_id	Optional[str]	ID of the room where the device is located
status	str	Current operational status ("on", "off", or custom states)
is_active	bool	Whether the device is currently available in the system

2.6 backend.models.energy_data

This module defines the EnergyData model used for storing energy consumption records

 $The \ model: \ - \ Represents \ energy \ consumption \ data \ for \ a \ specific \ device \ - \ Includes \ metadata \ such \ as \ timestamp \ \& \ location$

2.6.1 backend.models.energy data.EnergyData

Bases: BaseModel

Pydantic model representing energy consumption data for a smart device

Attributes:

Name	Туре	Description
device_id	str	Unique identifier for the device
timestamp	datetime	The date & time of the recorded energy consumption
energy_consumed	float	The amount of energy consumed in kilowatt-hours (kWh)
location	Optional[str]	The optional physical location of the device

2.7 backend.models.energy summary

This module defines the EnergySummary model for analyzing & reporting energy consumption

The EnergySummary model: - Aggregates energy consumption data over specified time periods - Supports reporting & historical data analysis requirements - Enables cost estimation based on energy usage - Facilitates comparisons between current & past usage periods - Provides the data foundation for energy-saving suggestions & insights

2.7.1 backend.models.energy_summary.EnergySummary

Bases: BaseModel

Pydantic model representing aggregated energy usage data for reporting

Provides summarized energy consumption data for specific time periods, enabling users to track trends & compare usage over time

Name	Туре	Description
user_id	str	ID of the user this summary belongs to
period	str	Time period of the summary ("daily", "weekly", "monthly")
start_date	datetime	Beginning of the summary period
end_date	datetime	End of the summary period
total_consumption	float	Total energy consumed in kWh during the period
cost_estimate	Optional[float]	Estimated cost of energy used
comparison_to_previous	Optional[float]	Percentage change compared to previous period

Source code in backend/models/energy_summary.py class EnergySummary(BaseModel): 16 17 18 Pydantic model representing aggregated energy usage data for reporting Provides summarized energy consumption data for specific time periods, enabling users to track trends & compare usage over time 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 Attributes: user_id (str): ID of the user this summary belongs to period (str): Time period of the summary ("daily", "weekly", "monthly") start_date (datetime): Beginning of the summary period end_date (datetime): End of the summary period total_consumption (float): Total energy consumed in kWh during the period cost_estimate (Optional[float]): Estimated cost of energy used comparison_to_previous (Optional[float]): Percentage change compared to previous period user id: str period: str # daily, weekly, monthly start_date: datetime total consumption: float cost_estimate: Optional[float] comparison_to_previous: Optional[float] # percentage change

2.8 backend.models.profile

This module defines the Profile model for managing user profiles in the Smart Family Energy Manager

The Profile model: - Represents different user types within the system (adult/elderly, child) - Stores accessibility preferences & personalization settings - Manages access control for features like device control & energy data - Supports the admin's ability to create & manage multiple profiles per household

2.8.1 backend.models.profile.Profile

Bases: BaseModel

Pydantic model representing a user profile in the Smart Family Energy Manager system

Profiles are created by an admin & can be customized for different family member with varying levels of access to system features & data

Name	Туре	Description
user_id	str	Unique identifier linking this profile to a user account
name	str	The display name for this profile
age	Optional[int]	The age of the profile user, used to tailor experience
profile_type	str	Type of profile - "adult", "elderly", or "child"
accessibility_settings	Dict	Dictionary containing accessibility preferences
can_control_devices	bool	Whether this profile can control smart devices
can_access_energy_data	bool	Whether this profile can view energy consumption data
can_manage_notifications	bool	Whether this profile can manage notification settings

2.9 backend.models.room

This module defines the Room model for organizing smart devices by location within the home

The Room model: - Provides a way to group devices by physical location - Supports the add/delete room functionality requirement - Enables room-based energy consumption analytics - Helps users organize & manage their smart home more efficiently - Facilitates location-specific automation rules & schedules

2.9.1 backend.models.room.Room

Bases: BaseModel

Pydantic model representing a room or area within the smart home

Rooms provide organizational structure for devices & help contextualize energy usage data by location

Name	Туре	Description	
id	str	Unique identifier for the room	
name	str	User-friendly name for the room (e.g., "Living Room", "Kitchen")	
created_by	str	ID of the user who created this room	

```
Class Room(BaseModel):

"""

Pydantic model representing a room or area within the smart home

Rooms provide organizational structure for devices & help contextualize
energy usage data by location

Attributes:
id (str): Unique identifier for the room
name (str): User-friendly name for the room (e.g., "Living Room", "Kitchen")
created_by (str): ID of the user who created this room

"""

id: str
name: str
created_by: str  # user_id
```

2.10 backend.models.schedule

This module defines the Schedule model for automating device operations within the smart home

The Schedule model: - Enables users to set timed operations for connected devices - Supports the scheduling & automation functional requirements - Allows for energy optimization through scheduled device usage - Provides time-based control of devices for convenience & efficiency - Implements the core functionality of device automation based on user-defined schedules

2.10.1 backend.models.schedule.Schedule

Bases: BaseModel

Pydantic model representing an automated schedule for a smart device

Allows users to create automated routines for devices to operate at specified times, supporting energy efficiency & convenience

Attributes:

Name	Туре	Description
device_id	str	ID of the device to be controlled by this schedule
start_time	datetime	Time when the scheduled operation should begin
end_time	datetime	Time when the scheduled operation should end
start_date	datetime	Date when the schedule becomes active
end_date	datetime	Date when the schedule expires
created_by	str	ID of the user who created this schedule
is_active	bool	Whether this schedule is currently enabled

2.11 backend.models.user

This model defines user-related data models for account creation & response handling

It includes: - UserCreate: A model for user registration input with email & password validation - UserResponse: A model for returning user details in API responses

2.11.1 backend.models.user.UserCreate

Bases: BaseModel

Pydantic model for user registration input

Attributes:

Name	Туре	Description
email	EmailStr	The user's email address
password	str	The user's password (validated for strength)
role	Optional[str]	The user's role, defaulting to "user"

```
Source code in backend/models/user.py
       class UserCreate(BaseModel):
13
14
              Pydantic model for user registration input
             email (EmailStr): The user's email address
password (str): The user's password (validated for strength)
role (Optional[str]): The user's role, defaulting to "user"
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
              email: EmailStr
              role: Optional[str] = "user"  # Default role
              @field validator("password")
               def validate_password(cls, value: str) -> str:
    """
                   Args:
                          value (str): The password provided by the user
                    Returns:
                          str: The validated password
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
                    ValueError: If the password doesn't meet security requirements
                   if len(value) < 8:
    raise ValueError("Password must be at least 8 characters long.")</pre>
                  if not any(char.isdigit() for char in value):
    raise ValueError("Password must contain at least 1 number.")
              raise ValueErro:("rassword must contain at least 1 letter.")

raise ValueErro("Password must contain at least 1 letter.")
              raise ValueError("Password must contain at if not any(char.islower() for char in value):
              if not any(char.islower() for char in value):
    raise ValueError("Password must contain at least 1 lower letter.")
if not any(char.isupper() for char in value):
    raise ValueError("Password must contain at least 1 upper letter.")
if not any(char in '!@#$%^s*()_+-=[]{}|;\':",.<>?/' for char in value):
    raise ValueError("Password must contain at least 1 special character.")
return value
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
              @field_validator("email")
               @classmethod
def validate_email(cls, value: EmailStr) -> EmailStr:
    """
                     Validate the user's email address
                          value (EmailStr): The email address provided
                    Returns:
                         EmailStr: The validated email
                    ValueError: If the email doesn't contain `@` or a valid domain
                    if not "@" in value:
                    raise ValueError("Email must contain @.")
if not "." in value.split("@")[1]:
                    raise ValueError("Email must contain a dot.")
return value
```

backend.models.user.UserCreate.validate email(value) classmethod

Validate the user's email address

Name	Туре	Description	Default
value	EmailStr	The email address provided	required

Returns:

Name	Туре	Description
EmailStr	EmailStr	The validated email

Raises:

Туре	Description	
ValueError	If the email doesn't contain @ or a valid domain	

 ${\tt backend.models.user.UserCreate.validate_password(value)} \quad {\tt classmethod}$

Validate the password to ensure it meets security requirements

Parameters:

Name	Туре	Description	Default
value	str	The password provided by the user	required

Returns:

Name	Туре	Description
str	str	The validated password

Туре	Description	
ValueError	If the password doesn't meet security requirements	

2.11.2 backend.models.user.UserResponse

Bases: BaseModel

Pydantic model for user response data

Name	Туре	Description
id	str	Unique identifier for the user
role	str	The assigned role of the user
email	EmailStr	The user's email address
is_verified	bool	Indicates whether the user's email address is verified (defaults to False)
created_at	Optional[datetime]	Timestamp of the user account creation

2.12 backend.db.database

This module handles MongoDB database connections & operations

It provides: - Initialization of MongoDB collections & indexes - Functions to fetch energy consumption data with optional filtering

2.12.1 backend.db.database.get energy data(start date=None, end date=None)

Fetch energy data consumption from MongoDB with optional date filtering

Parameters:

Name	Туре	Description	Default
start_date	str	Start date in YYYY-MM-DD format	None
end_date	str	End date in YYYY-MM-DD format	None

Returns:

Туре	Description	
List[Dict]	List[Dict]: A list of energy consumption records	

Raises:

Туре	Description
ValueError	If the provided date format is incorrect

```
Source code in backend/db/database.py
      def get_energy_data(start_date: Optional[str] = None, end_date: Optional[str] = None) -> List[Dict]:
           Fetch energy data consumption from MongoDB with optional date filtering
          Args:
start_date (str, optional): Start date in 'YYYY-MM-DD' format
end_date (str, optional): End date in 'YYYY-MM-DD' format
95
96
97
98
99
100
          Returns:
List[Dict]: A list of energy consumption records
          ValueError: If the provided date format is incorrect """
          query = {}
          if start date and end date:
104
105
               106
107
                   end_dt = datetime.strptime(end_date, "%Y-%m-%d")
query["timestamp"] = {"$gte": start_dt, "$lte": end_dt}
                  cept ValueError as exc:
raise ValueError("Invalid date format. Use `YYYY-MM-DD`.") from exc
          energy_data = list(energy_collection.find(query, {"_id": 0}))  # Exclude MongoDB _id field
           return energy_data
```

2.12.2 backend.db.database.init db() async

Initialize the database by creating necessary indexes

This function ensures: - Unique email constraint for users - Indexes on <code>device_id</code> & timestamp in the energy collection for optimized queries

Raises:

Туре	Description
RuntimeError	If there is an error during database initialization

```
Source code in backend/db/database.py
38
      async def init_db():
39
40
           Initialize the database by creating necessary indexes
41
42
           This function ensures:
- Unique email constraint for users
43
44
           - Indexes on `device_id` & `timestamp` in the energy collection for optimized queries
RuntimeError: If there is an error during database initialization
                # User collection indexes
                users_collection.create_index("email", unique = True)
               # Energy data collection index
               energy_collection.create_index("device_id")
energy_collection.create_index("timestamp")
               energy_collection.create index(("device id", 1), ("timestamp", 1)])
energy_collection.create_index("location")
              # Device collection indexes
devices_collection.create_index("id", unique=True)
devices_collection.create_index("room_id")
devices_collection.create_index("type")
                # Profile collection indexes
               profiles_collection.create_index("user_id")
profiles_collection.create_index([("user_id", 1), ("name", 1)], unique=True)
              rooms_collection.create_index("id", unique=True)
rooms_collection.create_index("created_by")
               schedules_collection.create_index("device_id")
schedules_collection.create_index("created_by"
               schedules collection.create index("start date")
               schedules_collection.create_index("end_date")
               # Energy summary collection indexes
summary_collection.create_index("user_id")
                81
82
                {\tt logger.info("Database\ initialized\ successfully\ with\ all\ collections\ and\ indexes.")}
83
84
                print("Database initialized successfully.")
              raise RuntimeError(f"Error during database initialization: {e}") from e
```

2.13 backend.core.config

2.14 backend.core.security

This module implements security features

Features include: - Password hashing - JWT token creation & verification - Role-based access control

2.14.1 backend.core.security.create_access_token(data, expires_delta=None)

Create a JWT token with an expiration time

Name	Туре	Description	Default
data	dict	The payload to encode in the token	required
expires_delta	Optional[timedelta]	The expiration time delta Defaults to ACCESS_TOKEN_EXPIRE_MINUTES	None

Returns:

Name	Туре	Description
str		The encoded JWT token

2.14.2 backend.core.security.get_current_user(token=Depends(oauth2_scheme))

Extract & validate the current user's JWT token

Parameters:

Name	Туре	Description	Default
token	str	The OAuth2 token obtained from authentication	Depends (oauth2_scheme)

Returns:

Name	Туре	Description
dict	dict	The decoded token payload

Туре	Description
HTTPException	If the token is invalid or missing required fields

2.14.3 backend.core.security.hash_password(password)

Hash a plain-text password using bcrypt

Parameters:

Name	Туре	Description	Default
password	str	The password to hash	required

Returns:

Name	Туре	Description
str	str	The hashed password

Raises:

Туре	Description
ValueError	If an error occurs during hashing

2.14.4 backend.core.security.needs rehash(hashed password)

Check if a stored hashed password required rehashing

Name	Туре	Description	Default
hashed_password	str	The existing hashed password	required

Returns:

Name	Туре	Description
bool	bool	True if rehashing is required & False if otherwise

```
def needs_rehash(hashed_password: str) -> bool:

"""
62
63
64
65
Args:
66
hashed_password (str): The existing hashed password
67
68
Returns:
69
bool: True if rehashing is required & False if otherwise
70
"""
71
return pwd_context.needs_update(hashed_password)
```

2.14.5 backend.core.security.role_required(required_role)

Dependency function to check to enforce role-based access control

Parameters:

Name	Туре	Description	Default
required_role	str	The required user role	required

Returns:

Name	Туре	Description
function		A dependency function that verifies the user's role

Туре	Description
HTTPException	If the user lacks the required role

```
Source code in backend/core/security.py
        def role_required(required_role: str):
128
129
            Dependency function to check to enforce role-based access control
                 required_role (str): The required user role
           Returns:
134
135
                function: A dependency function that verifies the user's role
136
137
            MAISES:
HTTPException: If the user lacks the required role
           Raises:
138
139
           def role_checker(current_user: dict = Depends(get_current_user)):
    if current_user.get("role") != required_role:
        raise HTTPException(
140
141
                         status_code = status.HTTP_403_FORBIDDEN,
detail = f"Permission denied: {required_role} role required"
145
146
            return role_checker
```

2.14.6 backend.core.security.verify_access_token(token)

Verify & decode a JWT access token

Parameters:

Name	Туре	Description	Default	
token	str	The JWT token to verify	required	

Returns:

Туре	Description
Optional[dict]	Optional[dict]: The decoded token payload if valid, otherwise None

```
Source code in backend/core/security.py

def verify_access_token(token: str) -> Optional[dict]:
    """

    Verify & decode a JWT access token

Args:
    token (str): The JWT token to verify

Returns:
    Optional[dict]: The decoded token payload if valid, otherwise None

"""

try:
    payload = jwt.decode(token, SECRET_KEY, algorithms = [ALGORITHM])
    return payload

except JWTError:
    return None
```

2.14.7 backend.core.security.verify_password(plain_password, hashed_password)

Verify a plain-text password against a hashed password

Parameters:

Name	Туре	Description	Default
plain_password	str	The input password	required
hashed_password	str	The stored hashed password	required

Returns:

Name	Туре	Description
bool	bool	True if the password matches & False if otherwise

Туре	Description
ValueError	If an error occurs during verification

```
def verify_password(plain_password: str, hashed_password: str) -> bool:

"""

Verify a plain-text password against a hashed password

Args:

Args:

Args:

Plain_password (str): The input password

hashed_password (str): The stored hashed password

Returns:

bool: True if the password matches & False if otherwise

Raises:

ValueError: If an error occurs during verification

"""

try:

return pwd_context.verify(plain_password, hashed_password)

except Exception as e:

raise ValueError(f"Error verifying password: {e}") from e
```