Sync Smart Home API Docs

None

Table of contents

1. Sync Smart Home API Documentation

2. API Reference

2.1 `app.main`

Main application entry point for the smart home API.

2.1.1 `app.main.root()` `async`

API health check endpoint.

> **Source code in** `app/main.py` ⌄
>
> ```
> 66    @app.get("/")
> 67    async def root():
> 68        """
> 69        API health check endpoint.
> 70        """
> 71        return {
> 72            "status": "online",
> 73            "message": "Sync Smart Home API is running",
> 74            "version": "1.0.0"
> 75        }
> ```

2.1.2 `app.main.startup_event()`

Initialize database on startup.

> **Source code in** `app/main.py` ⌄
>
> ```
> 41    @app.on_event("startup")
> 42    def startup_event():
> 43        """Initialize database on startup."""
> 44        init_db()
> 45
> 46        # Create reports directory if it doesn't exist
> 47        import os
> 48        from app.utils.report.report_generator import REPORTS_DIR
> 49        os.makedirs(REPORTS_DIR, exist_ok=True)
> ```

2.2 `app.core.auth`

Authentication dependencies for FastAPI.

2.2.1 `app.core.auth.get_current_user(credentials=Depends(security))` `async`

Get the currently authenticated user.

Uses HTTP Basic Authentication to validate the user.

Parameters:

Name	Type	Description	Default
`credentials`	`HTTPBasicCredentials`	The HTTP Basic credentials from the request	`Depends(security)`

Returns:

Name	Type	Description
UserDB	UserDB	The authenticated user

Raises:

Type	Description
HTTPException	If authentication fails

> **Source code in** `app/core/auth.py` ∨

```python
14    async def get_current_user(credentials: HTTPBasicCredentials = Depends(security)) -> UserDB:
15        """
16        Get the currently authenticated user.
17
18        Uses HTTP Basic Authentication to validate the user.
19
20        Args:
21            credentials: The HTTP Basic credentials from the request
22
23        Returns:
24            UserDB: The authenticated user
25
26        Raises:
27            HTTPException: If authentication fails
28        """
29        auth_exception = HTTPException(
30            status_code=status.HTTP_401_UNAUTHORIZED,
31            detail="Invalid username or password",
32            headers={"WWW-Authenticate": "Basic"},
33        )
34
35        # Try to find user by username
36        user = u_c.find_one({"username": credentials.username})
37
38        # If not found, try by email
39        if not user:
40            user = u_c.find_one({"email": credentials.username})
41
42        # Check credentials
43        if not user or not verify_password(credentials.password, user["hashed_password"]):
44            raise auth_exception
45
46        # Check if user is active
47        if not user.get("active", True):  # Default to active if not specified
48            raise HTTPException(
49                status_code=status.HTTP_401_UNAUTHORIZED,
50                detail="Inactive user account",
51                headers={"WWW-Authenticate": "Basic"},
52            )
53
54        return UserDB(**user)
```

2.3 `app.core.password`

Implements password security features.

2.3.1 `app.core.password.hash_password(p)`

Hashes a password.

Parameters:

Name	Type	Description	Default
p	str	Password to hash.	*required*

Returns:

Name	Type	Description
str	str	Hashed password.

Raises:

Type	Description
ValueError	If password fails to hash.

> **Source code in** `app/core/password.py` ⌄
>
> ```
>  9    def hash_password(p: str) -> str:
> 10        """
> 11        Hashes a password.
> 12
> 13        Args:
> 14            p (str): Password to hash.
> 15
> 16        Returns:
> 17            str: Hashed password.
> 18
> 19        Raises:
> 20            ValueError: If password fails to hash.
> 21        """
> 22        try:
> 23            return pc.hash(p)
> 24        except Exception as e:
> 25            raise ValueError(f"Error hashing password: {e}") from e
> ```

2.3.2 `app.core.password.verify_password(p, h)`

Matches plain-text password with the hashed password.

Parameters:

Name	Type	Description	Default
p	str	Plain-text password.	*required*
h	str	Hashed password.	*required*

Returns:

Name	Type	Description
bool	bool	(True: Passwords match); (False: Passwords don't match).

Raises:

Type	Description
ValueError	If passwords fail to compare.

> **Source code in** `app/core/password.py` ⌄

```
28   def verify_password(p: str, h: str) -> bool:
29       """
30       Matches plain-text password with the hashed password.
31
32       Args:
33           p (str): Plain-text password.
34           h (str): Hashed password.
35
36       Returns:
37           bool: (True: Passwords match); (False: Passwords don't match).
38
39       Raises:
40           ValueError: If passwords fail to compare.
41       """
42       try:
43           return pc.verify(p, h)
44       except Exception as e:
45           raise ValueError(f"Error verifying password: {e}") from e
```

2.3.3 `app.core.password.verify_role(u, r)`

Enforces correct user role for access.

Parameters:

Name	Type	Description	Default
u	str	User's current role.	*required*
r	str	Required user role.	*required*

Raises:

Type	Description
HTTPException	If user lacks the required role.

> **Source code in** `app/core/password.py` ⌄

```
47   def verify_role(u: str, r: str):
48       """
49       Enforces correct user role for access.
50
51       Args:
52           u (str): User's current role.
53           r (str): Required user role.
54
55       Raises:
56           HTTPException: If user lacks the required role.
57       """
58       if u != r:
59           raise HTTPException(
60               status_code=status.HTTP_403_FORBIDDEN,
61               detail=f"Permission denied. {r} role is required for access."
62           )
63
64       return True
```

2.4 `app.db.data`

Handles MongoDB database connections & operations.

2.4.1 `app.db.data.init_db()`

Initialize MongoDB database & creates indexes.

Source code in `app/db/data.py` ⌄

```python
31    def init_db():
32        """
33        Initialize MongoDB database & creates indexes.
34        """
35        # User collection
36        u_c.create_index("id", unique=True)      # Unique identification
37        u_c.create_index("email", unique=True)   # Unique email address
38        u_c.create_index("username")             # General username
39
40        # Profile collection
41        p_c.create_index("id", unique=True)          # Unique identification
42        p_c.create_index("user_id", unique=True)     # Unique user identification
43
44        # Device collection
45        d_c.create_index("id", unique=True)              # Device identification
46        d_c.create_index("user_id")                      # User identification
47        d_c.create_index([("type", 1), ("user_id", 1)]) # Filter type through user identification
48
49        # Room collection
50        r_c.create_index("id", unique=True)      # Unique identification
51        r_c.create_index("user_id")              # User identification
52        r_c.create_index("home_id")              # Home identification
53
54        # Usage collection
55        us_c.create_index("id", unique=True)                      # Unique identification
56        us_c.create_index("device_id")                            # Device identification
57        us_c.create_index("timestamp")                            # Usage log
58        us_c.create_index([("device_id", 1), ("timestamp", -1)])  # Filter log by device identification
59
60        # Automation collection
61        a_c.create_index("id", unique=True)      # Unique identification
62        a_c.create_index("user_id")              # User identification
63        a_c.create_index("device_id")            # User identification
64
65        # Notification collection
66        n_c.create_index("id", unique=True)                                # Unique identification
67        n_c.create_index("user_id")                                        # User identification
68        n_c.create_index([("user_id", 1), ("read", 1), ("timestamp", -1)]) # Filter notification read by device & time
69
70        # Access Management collection
71        am_c.create_index("id", unique=True)                        # Unique identification
72        am_c.create_index("owner_id")                               # Owner identification
73        am_c.create_index("resource_id")                            # Resource identification
74        am_c.create_index([("owner_id", 1), ("resource_id", 1)])    # Filters resource by its owner
75
76        # Goal collection
77        g_c.create_index("id", unique=True)                 # Unique identification
78        g_c.create_index("user_id")                         # User identification
79        g_c.create_index("type")                            # Type of goal
80        g_c.create_index([("user_id", 1), ("type", 1)])     # Filters types of goals by user identification
81
82        # Analytics collection
83        an_c.create_index("id", unique=True)                     # Unique identification
84        an_c.create_index("user_id")                             # User identification
85        an_c.create_index("device_id")                           # Device identification
86        an_c.create_index([("user_id", 1), ("timestamp", -1)])   # Filters user identification by timestamp
87
88        # Suggestion collection
89        s_c.create_index("id", unique=True)                                      # Unique identification
90        s_c.create_index("user_id")                                              # User identification
91        s_c.create_index([("user_id", 1), ("status", 1), ("timestamp", -1)])     # Filters user identification with status by timestamp
92
93        print("Database initialized with indexes.")
```

2.5 `app.models.access_management`

Models for access management validation.

2.5.1 `app.models.access_management.AccessLevel`

Bases: `str`, `Enum`

Enum for access permission levels.

> **Source code in** `app/models/access_management.py` ⌄
>
> ```
> 20    class AccessLevel(str, Enum):
> 21        """
> 22        Enum for access permission levels.
> 23        """
> 24        READ = "read"
> 25        CONTROL = "control"
> 26        MANAGE = "manage"
> 27        ADMIN = "admin"
> ```

2.5.2 `app.models.access_management.AccessManagementDB`

Bases: `BaseModel`

Internal model representing access management data in the database.

Attributes:

Name	Type	Description
id	str	Unique identifier.
owner_id	str	ID of the user who owns the resource.
resource_id	str	ID of the resource being shared.
resource_type	ResourceType	Type of resource being shared.
user_id	str	ID of the user granted access.
access_level	AccessLevel	Level of access granted.
created	datetime	When the access was granted.
updated	Optional[datetime]	When the access was last updated.
expires_at	Optional[datetime]	When the access expires.
active	bool	Whether this access grant is currently active.
note	Optional[str]	Optional note about this access grant.

Source code in `app/models/access_management.py` ⌄

```
93    class AccessManagementDB(BaseModel):
94        """
95        Internal model representing access management data in the database.
96
97        Attributes:
98            id (str): Unique identifier.
99            owner_id (str): ID of the user who owns the resource.
100           resource_id (str): ID of the resource being shared.
101           resource_type (ResourceType): Type of resource being shared.
102           user_id (str): ID of the user granted access.
103           access_level (AccessLevel): Level of access granted.
104           created (datetime): When the access was granted.
105           updated (Optional[datetime]): When the access was last updated.
106           expires_at (Optional[datetime]): When the access expires.
107           active (bool): Whether this access grant is currently active.
108           note (Optional[str]): Optional note about this access grant.
109       """
110       id: str = Field(default_factory=lambda: str(uuid.uuid4()))
111       owner_id: str
112       resource_id: str
113       resource_type: ResourceType
114       user_id: str
115       access_level: AccessLevel
116       created: datetime = Field(default_factory=datetime.utcnow)
117       updated: Optional[datetime] = None
118       expires_at: Optional[datetime] = None
119       active: bool = True
120       note: Optional[str] = None
121
122       model_config = ConfigDict(from_attributes=True)
```

2.5.3 `app.models.access_management.AccessManagementResponse`

Bases: `BaseModel`

Model for access management data returned in API responses.

Attributes:

Name	Type	Description
id	str	Unique identifier.
owner_id	str	ID of the user who owns the resource.
resource_id	str	ID of the resource being shared.
resource_type	ResourceType	Type of resource being shared.
user_id	str	ID of the user granted access.
access_level	AccessLevel	Level of access granted.
created	datetime	When the access was granted.
expires_at	Optional[datetime]	When the access expires.
active	bool	Whether this access grant is currently active.

Source code in `app/models/access_management.py` ⌄

```
125    class AccessManagementResponse(BaseModel):
126        """
127        Model for access management data returned in API responses.
128
129        Attributes:
130            id (str): Unique identifier.
131            owner_id (str): ID of the user who owns the resource.
132            resource_id (str): ID of the resource being shared.
133            resource_type (ResourceType): Type of resource being shared.
134            user_id (str): ID of the user granted access.
135            access_level (AccessLevel): Level of access granted.
136            created (datetime): When the access was granted.
137            expires_at (Optional[datetime]): When the access expires.
138            active (bool): Whether this access grant is currently active.
139        """
140        id: str
141        owner_id: str
142        resource_id: str
143        resource_type: ResourceType
144        user_id: str
145        access_level: AccessLevel
146        created: datetime
147        expires_at: Optional[datetime] = None
148        active: bool
149        note: Optional[str] = None
150
151        model_config = ConfigDict(from_attributes=True)
```

2.5.4 `app.models.access_management.AccessManagementUpdate`

Bases: `BaseModel`

Model for updating access management entries.

Attributes:

Name	Type	Description
access_level	Optional[AccessLevel]	Updated access level.
expires_at	Optional[datetime]	Updated expiration time.
active	Optional[bool]	Updated active status.
note	Optional[str]	Updated note.

> ## Source code in `app/models/access_management.py` ⌄

```
154   class AccessManagementUpdate(BaseModel):
155       """
156       Model for updating access management entries.
157
158       Attributes:
159           access_level (Optional[AccessLevel]): Updated access level.
160           expires_at (Optional[datetime]): Updated expiration time.
161           active (Optional[bool]): Updated active status.
162           note (Optional[str]): Updated note.
163       """
164       access_level: Optional[AccessLevel] = None
165       expires_at: Optional[datetime] = None
166       active: Optional[bool] = None
167       note: Optional[str] = None
168
169       @field_validator("note")
170       @classmethod
171       def validate_note(cls, n: Optional[str]) -> Optional[str]:
172           """
173           Validate note length.
174
175           Arguments:
176               n (Optional[str]): Note to validate.
177
178           Returns:
179               Optional[str]: Validated note.
180
181           Raises:
182               ValueError: If validation fails.
183           """
184           if n is not None and len(n) > 200:
185               raise ValueError("Note must be less than 200 characters long.")
186
187           return n
```

`app.models.access_management.AccessManagementUpdate.validate_note(n)` `classmethod`

Validate note length.

Parameters:

Name	Type	Description	Default
n	Optional[str]	Note to validate.	*required*

Returns:

Type	Description
Optional[str]	Optional[str]: Validated note.

Raises:

Type	Description
ValueError	If validation fails.

Source code in `app/models/access_management.py` ∨

```
169    @field_validator("note")
170    @classmethod
171    def validate_note(cls, n: Optional[str]) -> Optional[str]:
172        """
173        Validate note length.
174
175        Arguments:
176            n (Optional[str]): Note to validate.
177
178        Returns:
179            Optional[str]: Validated note.
180
181        Raises:
182            ValueError: If validation fails.
183        """
184        if n is not None and len(n) > 200:
185            raise ValueError("Note must be less than 200 characters long.")
186
187        return n
```

2.5.5 `app.models.access_management.CreateAccessManagement`

Bases: `BaseModel`

Model for creating access management entries.

Attributes:

Name	Type	Description
owner_id	str	ID of the user who owns the resource.
resource_id	str	ID of the resource being shared.
resource_type	ResourceType	Type of resource being shared.
user_ids	List[str]	List of user IDs to grant access to.
access_level	AccessLevel	Level of access to grant.
expires_at	Optional[datetime]	When the access expires (optional).
note	Optional[str]	Optional note about this access grant.

Source code in `app/models/access_management.py` ⌄

```
29    class CreateAccessManagement(BaseModel):
30        """
31        Model for creating access management entries.
32
33        Attributes:
34            owner_id (str): ID of the user who owns the resource.
35            resource_id (str): ID of the resource being shared.
36            resource_type (ResourceType): Type of resource being shared.
37            user_ids (List[str]): List of user IDs to grant access to.
38            access_level (AccessLevel): Level of access to grant.
39            expires_at (Optional[datetime]): When the access expires (optional).
40            note (Optional[str]): Optional note about this access grant.
41        """
42        owner_id: str
43        resource_id: str
44        resource_type: ResourceType
45        user_ids: List[str]
46        access_level: AccessLevel
47        expires_at: Optional[datetime] = None
48        note: Optional[str] = None
49
50        @field_validator("note")
51        @classmethod
52        def validate_note(cls, n: Optional[str]) -> Optional[str]:
53            """
54            Validate note length.
55
56            Arguments:
57                n (Optional[str]): Note to validate.
58
59            Returns:
60                Optional[str]: Validated note.
61
62            Raises:
63                ValueError: If validation fails.
64            """
65            if n is not None and len(n) > 200:
66                raise ValueError("Note must be less than 200 characters long.")
67
68            return n
69
70        @field_validator("user_ids")
71        @classmethod
72        def validate_user_ids(cls, u: List[str]) -> List[str]:
73            """
74            Validate user IDs list.
75
76            Arguments:
77                u (List[str]): List of user IDs.
78
79            Returns:
80                List[str]: Validated list of user IDs.
81
82            Raises:
83                ValueError: If validation fails.
84            """
85            if not u:
86                raise ValueError("At least one user ID must be provided.")
87            if len(u) > 50:
88                raise ValueError("Cannot share with more than 50 users at once.")
89
90            return u
```

`app.models.access_management.CreateAccessManagement.validate_note(n)` `classmethod`

Validate note length.

Parameters:

Name	Type	Description	Default
n	Optional[str]	Note to validate.	*required*

Returns:

Type	Description
Optional[str]	Optional[str]: Validated note.

Raises:

Type	Description
ValueError	If validation fails.

Source code in `app/models/access_management.py` ⌄

```
50    @field_validator("note")
51    @classmethod
52    def validate_note(cls, n: Optional[str]) -> Optional[str]:
53        """
54        Validate note length.
55
56        Arguments:
57            n (Optional[str]): Note to validate.
58
59        Returns:
60            Optional[str]: Validated note.
61
62        Raises:
63            ValueError: If validation fails.
64        """
65        if n is not None and len(n) > 200:
66            raise ValueError("Note must be less than 200 characters long.")
67
68        return n
```

app.models.access_management.CreateAccessManagement.validate_user_ids(u) `classmethod`

Validate user IDs list.

Parameters:

Name	Type	Description	Default
u	List[str]	List of user IDs.	*required*

Returns:

Type	Description
List[str]	List[str]: Validated list of user IDs.

Raises:

Type	Description
ValueError	If validation fails.

Source code in `app/models/access_management.py` ⌄

```
70   @field_validator("user_ids")
71   @classmethod
72   def validate_user_ids(cls, u: List[str]) -> List[str]:
73       """
74       Validate user IDs list.
75
76       Arguments:
77           u (List[str]): List of user IDs.
78
79       Returns:
80           List[str]: Validated list of user IDs.
81
82       Raises:
83           ValueError: If validation fails.
84       """
85       if not u:
86           raise ValueError("At least one user ID must be provided.")
87       if len(u) > 50:
88           raise ValueError("Cannot share with more than 50 users at once.")
89
90       return u
```

2.5.6 `app.models.access_management.ResourceType`

Bases: `str`, `Enum`

Enum for supported resource types in the system.

Source code in `app/models/access_management.py` ⌄

```
10   class ResourceType(str, Enum):
11       """
12       Enum for supported resource types in the system.
13       """
14       DEVICE = "device"
15       ROOM = "room"
16       HOME = "home"
17       AUTOMATION = "automation"
```

2.6 `app.models.analytics`

Models for analytics validation and storage.

2.6.1 `app.models.analytics.AnalyticsDB`

Bases: `BaseModel`

Internal model representing analytics data in the database.

Attributes:

Name	Type	Description
id	str	Unique analytics identifier.
user_id	str	ID of the user the analytics belongs to.
device_id	str	ID of the device generating the analytics.
data_type	str	Type of analytics data.
metrics	Dict[str, Any]	The actual metrics being stored.
tags	List[str]	Tags for categorizing analytics data.
timestamp	datetime	When the analytics data was recorded.
updated	Optional[datetime]	When the analytics data was last updated.

Source code in `app/models/analytics.py`

```
79    class AnalyticsDB(BaseModel):
80        """
81        Internal model representing analytics data in the database.
82
83        Attributes:
84            id (str): Unique analytics identifier.
85            user_id (str): ID of the user the analytics belongs to.
86            device_id (str): ID of the device generating the analytics.
87            data_type (str): Type of analytics data.
88            metrics (Dict[str, Any]): The actual metrics being stored.
89            tags (List[str]): Tags for categorizing analytics data.
90            timestamp (datetime): When the analytics data was recorded.
91            updated (Optional[datetime]): When the analytics data was last updated.
92        """
93        id: str = Field(default_factory=lambda: str(uuid.uuid4()))
94        user_id: str
95        device_id: str
96        data_type: str
97        metrics: Dict[str, Any]
98        tags: List[str] = []
99        timestamp: datetime = Field(default_factory=datetime.utcnow)
100       updated: Optional[datetime] = None
101
102       model_config = ConfigDict(from_attributes=True)
```

2.6.2 `app.models.analytics.AnalyticsQuery`

Bases: `BaseModel`

Model for querying analytics data with filters.

Attributes:

Name	Type	Description
user_id	Optional[str]	Filter by user ID.
device_id	Optional[str]	Filter by device ID.
data_type	Optional[str]	Filter by data type.
start_time	Optional[datetime]	Filter by start timestamp.
end_time	Optional[datetime]	Filter by end timestamp.
tags	Optional[List[str]]	Filter by tags.

Source code in `app/models/analytics.py` ⌄

```
165    class AnalyticsQuery(BaseModel):
166        """
167        Model for querying analytics data with filters.
168
169        Attributes:
170            user_id (Optional[str]): Filter by user ID.
171            device_id (Optional[str]): Filter by device ID.
172            data_type (Optional[str]): Filter by data type.
173            start_time (Optional[datetime]): Filter by start timestamp.
174            end_time (Optional[datetime]): Filter by end timestamp.
175            tags (Optional[List[str]]): Filter by tags.
176        """
177        user_id: Optional[str] = None
178        device_id: Optional[str] = None
179        data_type: Optional[str] = None
180        start_time: Optional[datetime] = None
181        end_time: Optional[datetime] = None
182        tags: Optional[List[str]] = None
183
184        @field_validator("start_time", "end_time")
185        @classmethod
186        def validate_time_range(cls, v: Optional[datetime], info) -> Optional[datetime]:
187            """
188            Validate that start_time comes before end_time if both are provided.
189
190            Note: This validation only runs after all fields are populated, so it
191            needs to be called separately after instantiation.
192            """
193            return v
194
195        def validate_time_range_post_init(self):
196            """
197            Validate that start_time comes before end_time if both are provided.
198            Call this after instantiating the model.
199            """
200            if self.start_time and self.end_time and self.start_time > self.end_time:
201                raise ValueError("start_time must be before end_time")
```

`app.models.analytics.AnalyticsQuery.validate_time_range(v, info)` `classmethod`

Validate that start_time comes before end_time if both are provided.

Note: This validation only runs after all fields are populated, so it needs to be called separately after instantiation.

Source code in `app/models/analytics.py` ⌄

```
184        @field_validator("start_time", "end_time")
185        @classmethod
186        def validate_time_range(cls, v: Optional[datetime], info) -> Optional[datetime]:
187            """
188            Validate that start_time comes before end_time if both are provided.
189
190            Note: This validation only runs after all fields are populated, so it
191            needs to be called separately after instantiation.
192            """
193            return v
```

`app.models.analytics.AnalyticsQuery.validate_time_range_post_init()`

Validate that start_time comes before end_time if both are provided. Call this after instantiating the model.

Source code in `app/models/analytics.py` ⌄

```
195        def validate_time_range_post_init(self):
196            """
197            Validate that start_time comes before end_time if both are provided.
198            Call this after instantiating the model.
199            """
200            if self.start_time and self.end_time and self.start_time > self.end_time:
201                raise ValueError("start_time must be before end_time")
```

2.6.3 `app.models.analytics.AnalyticsResponse`

Bases: `BaseModel`

Model for analytics data returned in API responses.

Attributes:

Name	Type	Description
id	str	Unique analytics identifier.
user_id	str	ID of the user the analytics belongs to.
device_id	str	ID of the device generating the analytics.
data_type	str	Type of analytics data.
metrics	Dict[str, Any]	The actual metrics being stored.
tags	List[str]	Tags for categorizing analytics data.
timestamp	datetime	When the analytics data was recorded.

Source code in `app/models/analytics.py` ⌄

```python
106    class AnalyticsResponse(BaseModel):
107        """
108        Model for analytics data returned in API responses.
109
110        Attributes:
111            id (str): Unique analytics identifier.
112            user_id (str): ID of the user the analytics belongs to.
113            device_id (str): ID of the device generating the analytics.
114            data_type (str): Type of analytics data.
115            metrics (Dict[str, Any]): The actual metrics being stored.
116            tags (List[str]): Tags for categorizing analytics data.
117            timestamp (datetime): When the analytics data was recorded.
118        """
119        id: str
120        user_id: str
121        device_id: str
122        data_type: str
123        metrics: Dict[str, Any]
124        tags: List[str]
125        timestamp: datetime
126
127        model_config = ConfigDict(from_attributes=True)
```

2.6.4 `app.models.analytics.AnalyticsUpdate`

Bases: `BaseModel`

Model for updating analytics information.

Attributes:

Name	Type	Description
metrics	Optional[Dict[str, Any]]	Updated metrics.
tags	Optional[List[str]]	Updated tags for categorizing analytics data.

❞ Source code in `app/models/analytics.py` ⌄

```
131    class AnalyticsUpdate(BaseModel):
132        """
133        Model for updating analytics information.
134
135        Attributes:
136            metrics (Optional[Dict[str, Any]]): Updated metrics.
137            tags (Optional[List[str]]): Updated tags for categorizing analytics data.
138        """
139        metrics: Optional[Dict[str, Any]] = None
140        tags: Optional[List[str]] = None
141
142        @field_validator("tags")
143        @classmethod
144        def validate_tags(cls, v: Optional[List[str]]) -> Optional[List[str]]:
145            """
146            Validate tags.
147
148            Arguments:
149                v (Optional[List[str]]): Tags to be validated.
150
151            Returns:
152                Optional[List[str]]: Validated tags.
153
154            Raises:
155                ValueError: If tags contain empty strings.
156            """
157            if v is not None:
158                if any(not tag.strip() for tag in v):
159                    raise ValueError("Tags cannot contain empty strings.")
160                return [tag.strip() for tag in v]
161            return v
```

`app.models.analytics.AnalyticsUpdate.validate_tags(v)` `classmethod`

Validate tags.

Parameters:

Name	Type	Description	Default
v	Optional[List[str]]	Tags to be validated.	*required*

Returns:

Type	Description
Optional[List[str]]	Optional[List[str]]: Validated tags.

Raises:

Type	Description
ValueError	If tags contain empty strings.

❝ Source code in `app/models/analytics.py` ⌄

```
142    @field_validator("tags")
143    @classmethod
144    def validate_tags(cls, v: Optional[List[str]]) -> Optional[List[str]]:
145        """
146        Validate tags.
147
148        Arguments:
149            v (Optional[List[str]]): Tags to be validated.
150
151        Returns:
152            Optional[List[str]]: Validated tags.
153
154        Raises:
155            ValueError: If tags contain empty strings.
156        """
157        if v is not None:
158            if any(not tag.strip() for tag in v):
159                raise ValueError("Tags cannot contain empty strings.")
160            return [tag.strip() for tag in v]
161        return v
```

2.6.5 `app.models.analytics.CreateAnalytics`

Bases: `BaseModel`

Model for analytics data input.

Attributes:

Name	Type	Description
user_id	str	ID of the user the analytics belongs to.
device_id	str	ID of the device generating the analytics.
data_type	str	Type of analytics data (energy, usage, temperature, etc.).
metrics	Dict[str, Any]	The actual metrics being stored.
tags	List[str]	Optional tags for categorizing analytics data.

Source code in `app/models/analytics.py` ⌄

```
10   class CreateAnalytics(BaseModel):
11       """
12       Model for analytics data input.
13
14       Attributes:
15           user_id (str): ID of the user the analytics belongs to.
16           device_id (str): ID of the device generating the analytics.
17           data_type (str): Type of analytics data (energy, usage, temperature, etc.).
18           metrics (Dict[str, Any]): The actual metrics being stored.
19           tags (List[str]): Optional tags for categorizing analytics data.
20       """
21       user_id: str
22       device_id: str
23       data_type: str
24       metrics: Dict[str, Any]
25       tags: Optional[List[str]] = []
26
27       @field_validator("user_id", "device_id")
28       @classmethod
29       def validate_id(cls, v: str) -> str:
30           """
31           Validate ID fields.
32
33           Arguments:
34               v (str): ID to be validated.
35
36           Returns:
37               str: Validated ID.
38
39           Raises:
40               ValueError: If ID is empty or doesn't match UUID format.
41           """
42           if not v:
43               raise ValueError("ID cannot be empty.")
44
45           # Try to parse as UUID to ensure valid format
46           try:
47               uuid.UUID(v)
48           except ValueError as exc:
49               raise ValueError("ID must be a valid UUID format.") from exc
50
51           return v
52
53       @field_validator("data_type")
54       @classmethod
55       def validate_data_type(cls, v: str) -> str:
56           """
57           Validate data_type field.
58
59           Arguments:
60               v (str): data_type to be validated.
61
62           Returns:
63               str: Validated data_type.
64
65           Raises:
66               ValueError: If data_type is empty or invalid.
67           """
68           valid_types = ["energy", "usage", "temperature", "humidity", "motion", "light", "other"]
69           if not v:
70               raise ValueError("Data type cannot be empty.")
71
72           if v.lower() not in valid_types:
73               raise ValueError(f"Data type must be one of: {', '.join(valid_types)}")
74
75           return v.lower()
```

`app.models.analytics.CreateAnalytics.validate_data_type(v)` `classmethod`

Validate data_type field.

Parameters:

Name	Type	Description	Default
v	str	data_type to be validated.	*required*

Returns:

Name	Type	Description
str	str	Validated data_type.

Raises:

Type	Description
ValueError	If data_type is empty or invalid.

Source code in `app/models/analytics.py` ˅

```
53   @field_validator("data_type")
54   @classmethod
55   def validate_data_type(cls, v: str) -> str:
56       """
57       Validate data_type field.
58
59       Arguments:
60           v (str): data_type to be validated.
61
62       Returns:
63           str: Validated data_type.
64
65       Raises:
66           ValueError: If data_type is empty or invalid.
67       """
68       valid_types = ["energy", "usage", "temperature", "humidity", "motion", "light", "other"]
69       if not v:
70           raise ValueError("Data type cannot be empty.")
71
72       if v.lower() not in valid_types:
73           raise ValueError(f"Data type must be one of: {', '.join(valid_types)}")
74
75       return v.lower()
```

`app.models.analytics.CreateAnalytics.validate_id(v)` classmethod

Validate ID fields.

Parameters:

Name	Type	Description	Default
v	str	ID to be validated.	*required*

Returns:

Name	Type	Description
str	str	Validated ID.

Raises:

Type	Description
ValueError	If ID is empty or doesn't match UUID format.

> **Source code in** `app/models/analytics.py` ⌄

```
27    @field_validator("user_id", "device_id")
28    @classmethod
29    def validate_id(cls, v: str) -> str:
30        """
31        Validate ID fields.
32
33        Arguments:
34            v (str): ID to be validated.
35
36        Returns:
37            str: Validated ID.
38
39        Raises:
40            ValueError: If ID is empty or doesn't match UUID format.
41        """
42        if not v:
43            raise ValueError("ID cannot be empty.")
44
45        # Try to parse as UUID to ensure valid format
46        try:
47            uuid.UUID(v)
48        except ValueError as exc:
49            raise ValueError("ID must be a valid UUID format.") from exc
50
51        return v
```

2.7 `app.models.automation`

Models for automation validation.

2.7.1 `app.models.automation.ActionType`

Bases: `str`, `Enum`

Enum for different types of automation actions.

> **Source code in** `app/models/automation.py` ⌄

```
21    class ActionType(str, Enum):
22        """
23        Enum for different types of automation actions.
24        """
25        DEVICE_CONTROL = "device_control"
26        NOTIFICATION = "notification"
27        SCENE_ACTIVATION = "scene_activation"
28        ENERGY_MANAGEMENT = "energy_management"
```

2.7.2 `app.models.automation.AutomationDB`

Bases: `BaseModel`

Internal model representing automation data in the database.

Attributes:

Name	Type	Description
id	str	Unique automation identifier.
name	str	Name of the automation.
description	str	Description of what the automation does.
user_id	str	ID of the user who owns this automation.
device_id	str	ID of the device associated with this automation.
enabled	bool	Whether the automation is enabled.
trigger_type	TriggerType	Type of trigger for this automation.
trigger_data	Dict	Configuration data for the trigger.
action_type	ActionType	Type of action for this automation.
action_data	Dict	Configuration data for the action.
conditions	Optional[List[Dict]]	Optional conditions that must be met.
created	datetime	When the automation was created.
updated	Optional[datetime]	When the automation was last updated.
last_triggered	Optional[datetime]	When the automation was last triggered.
execution_count	int	Number of times the automation has executed.

Source code in `app/models/automation.py`

```python
101    class AutomationDB(BaseModel):
102        """
103        Internal model representing automation data in the database.
104
105        Attributes:
106            id (str): Unique automation identifier.
107            name (str): Name of the automation.
108            description (str): Description of what the automation does.
109            user_id (str): ID of the user who owns this automation.
110            device_id (str): ID of the device associated with this automation.
111            enabled (bool): Whether the automation is enabled.
112            trigger_type (TriggerType): Type of trigger for this automation.
113            trigger_data (Dict): Configuration data for the trigger.
114            action_type (ActionType): Type of action for this automation.
115            action_data (Dict): Configuration data for the action.
116            conditions (Optional[List[Dict]]): Optional conditions that must be met.
117            created (datetime): When the automation was created.
118            updated (Optional[datetime]): When the automation was last updated.
119            last_triggered (Optional[datetime]): When the automation was last triggered.
120            execution_count (int): Number of times the automation has executed.
121        """
122        id: str = Field(default_factory=lambda: str(uuid.uuid4()))
123        name: str
124        description: str
125        user_id: str
126        device_id: str
127        enabled: bool = True
128        trigger_type: TriggerType
129        trigger_data: Dict[str, Any]
130        action_type: ActionType
131        action_data: Dict[str, Any]
132        conditions: Optional[List[Dict[str, Any]]] = None
133        created: datetime = Field(default_factory=datetime.utcnow)
134        updated: Optional[datetime] = None
135        last_triggered: Optional[datetime] = None
136        execution_count: int = 0
137
138        model_config = ConfigDict(from_attributes=True)
```

2.7.3 `app.models.automation.AutomationDetailResponse`

Bases: `AutomationResponse`

Detailed model for automation data returned in API responses. Extends AutomationResponse to include configuration details.

> **dditional Attributes** ∨
>
> trigger_data (Dict): Configuration data for the trigger. action_data (Dict): Configuration data for the action. conditions (Optional[List[Dict]]): Optional conditions that must be met. updated (Optional[datetime]): When the automation was last updated.

> **Source code in** `app/models/automation.py` ∨
>
> ```python
> 175    class AutomationDetailResponse(AutomationResponse):
> 176        """
> 177        Detailed model for automation data returned in API responses.
> 178        Extends AutomationResponse to include configuration details.
> 179
> 180        Additional Attributes:
> 181            trigger_data (Dict): Configuration data for the trigger.
> 182            action_data (Dict): Configuration data for the action.
> 183            conditions (Optional[List[Dict]]): Optional conditions that must be met.
> 184            updated (Optional[datetime]): When the automation was last updated.
> 185        """
> 186        trigger_data: Dict[str, Any]
> 187        action_data: Dict[str, Any]
> 188        conditions: Optional[List[Dict[str, Any]]] = None
> 189        updated: Optional[datetime] = None
> ```

2.7.4 `app.models.automation.AutomationResponse`

Bases: `BaseModel`

Model for automation data returned in API responses.

Attributes:

Name	Type	Description
id	str	Unique automation identifier.
name	str	Name of the automation.
description	str	Description of what the automation does.
user_id	str	ID of the user who owns this automation.
device_id	str	ID of the device associated with this automation.
enabled	bool	Whether the automation is enabled.
trigger_type	TriggerType	Type of trigger for this automation.
action_type	ActionType	Type of action for this automation.
created	datetime	When the automation was created.
last_triggered	Optional[datetime]	When the automation was last triggered.
execution_count	int	Number of times the automation has executed.

> **Source code in** `app/models/automation.py` ⌄

```
142    class AutomationResponse(BaseModel):
143        """
144        Model for automation data returned in API responses.
145
146        Attributes:
147            id (str): Unique automation identifier.
148            name (str): Name of the automation.
149            description (str): Description of what the automation does.
150            user_id (str): ID of the user who owns this automation.
151            device_id (str): ID of the device associated with this automation.
152            enabled (bool): Whether the automation is enabled.
153            trigger_type (TriggerType): Type of trigger for this automation.
154            action_type (ActionType): Type of action for this automation.
155            created (datetime): When the automation was created.
156            last_triggered (Optional[datetime]): When the automation was last triggered.
157            execution_count (int): Number of times the automation has executed.
158        """
159        id: str
160        name: str
161        description: str
162        user_id: str
163        device_id: str
164        enabled: bool
165        trigger_type: TriggerType
166        action_type: ActionType
167        created: datetime
168        last_triggered: Optional[datetime] = None
169        execution_count: int
170
171        model_config = ConfigDict(from_attributes=True)
```

2.7.5 `app.models.automation.AutomationUpdate`

Bases: `BaseModel`

Model for updating automation information.

Attributes:

Name	Type	Description
name	Optional[str]	Updated name of the automation.
description	Optional[str]	Updated description.
enabled	Optional[bool]	Updated enabled status.
trigger_type	Optional[TriggerType]	Updated trigger type.
trigger_data	Optional[Dict]	Updated trigger configuration.
action_type	Optional[ActionType]	Updated action type.
action_data	Optional[Dict]	Updated action configuration.
conditions	Optional[List[Dict]]	Updated conditions.

Source code in `app/models/automation.py` ⌄

```
193    class AutomationUpdate(BaseModel):
194        """
195        Model for updating automation information.
196
197        Attributes:
198            name (Optional[str]): Updated name of the automation.
199            description (Optional[str]): Updated description.
200            enabled (Optional[bool]): Updated enabled status.
201            trigger_type (Optional[TriggerType]): Updated trigger type.
202            trigger_data (Optional[Dict]): Updated trigger configuration.
203            action_type (Optional[ActionType]): Updated action type.
204            action_data (Optional[Dict]): Updated action configuration.
205            conditions (Optional[List[Dict]]): Updated conditions.
206        """
207        name: Optional[str] = None
208        description: Optional[str] = None
209        enabled: Optional[bool] = None
210        trigger_type: Optional[TriggerType] = None
211        trigger_data: Optional[Dict[str, Any]] = None
212        action_type: Optional[ActionType] = None
213        action_data: Optional[Dict[str, Any]] = None
214        conditions: Optional[List[Dict[str, Any]]] = None
215
216        @field_validator("name")
217        @classmethod
218        def validate_name(cls, v: Optional[str]) -> Optional[str]:
219            """
220            Validate automation name.
221
222            Arguments:
223                v (Optional[str]): Name to be validated.
224
225            Returns:
226                Optional[str]: Validated name.
227
228            Raises:
229                ValueError: Validation encountered a missing requirement.
230            """
231            if isinstance(v, str):
232                if len(v) < 3:
233                    raise ValueError("Automation name must be at least 3 characters long.")
234                if len(v) > 50:
235                    raise ValueError("Automation name must be less than 50 characters long.")
236
237            return v
238
239        @field_validator("description")
240        @classmethod
241        def validate_description(cls, v: Optional[str]) -> Optional[str]:
242            """
243            Validate automation description.
244
245            Arguments:
246                v (Optional[str]): Description to be validated.
247
248            Returns:
249                Optional[str]: Validated description.
250
251            Raises:
252                ValueError: Validation encountered a missing requirement.
253            """
254            if isinstance(v, str) and len(v) > 500:
255                raise ValueError("Automation description must be less than 500 characters long.")
256
257            return v
```

`app.models.automation.AutomationUpdate.validate_description(v)` `classmethod`

Validate automation description.

Parameters:

Name	Type	Description	Default
v	Optional[str]	Description to be validated.	*required*

Returns:

Type	Description
Optional[str]	Optional[str]: Validated description.

Raises:

Type	Description
`ValueError`	Validation encountered a missing requirement.

> **Source code in** `app/models/automation.py` ⌄
>
> ```
> 239    @field_validator("description")
> 240    @classmethod
> 241    def validate_description(cls, v: Optional[str]) -> Optional[str]:
> 242        """
> 243        Validate automation description.
> 244
> 245        Arguments:
> 246            v (Optional[str]): Description to be validated.
> 247
> 248        Returns:
> 249            Optional[str]: Validated description.
> 250
> 251        Raises:
> 252            ValueError: Validation encountered a missing requirement.
> 253        """
> 254        if isinstance(v, str) and len(v) > 500:
> 255            raise ValueError("Automation description must be less than 500 characters long.")
> 256
> 257        return v
> ```

`app.models.automation.AutomationUpdate.validate_name(v)` `classmethod`

Validate automation name.

Parameters:

Name	Type	Description	Default
`v`	`Optional[str]`	Name to be validated.	*required*

Returns:

Type	Description
`Optional[str]`	Optional[str]: Validated name.

Raises:

Type	Description
`ValueError`	Validation encountered a missing requirement.

> **Source code in** `app/models/automation.py` ⌄

```
216    @field_validator("name")
217    @classmethod
218    def validate_name(cls, v: Optional[str]) -> Optional[str]:
219        """
220        Validate automation name.
221
222        Arguments:
223            v (Optional[str]): Name to be validated.
224
225        Returns:
226            Optional[str]: Validated name.
227
228        Raises:
229            ValueError: Validation encountered a missing requirement.
230        """
231        if isinstance(v, str):
232            if len(v) < 3:
233                raise ValueError("Automation name must be at least 3 characters long.")
234            if len(v) > 50:
235                raise ValueError("Automation name must be less than 50 characters long.")
236
237        return v
```

2.7.6 `app.models.automation.CreateAutomation`

Bases: `BaseModel`

Model for automation creation input.

Attributes:

Name	Type	Description
name	str	Name of the automation.
description	str	Description of what the automation does.
user_id	str	ID of the user who owns this automation.
device_id	str	ID of the device associated with this automation.
enabled	bool	Whether the automation is enabled.
trigger_type	TriggerType	Type of trigger for this automation.
trigger_data	Dict	Configuration data for the trigger.
action_type	ActionType	Type of action for this automation.
action_data	Dict	Configuration data for the action.
conditions	Optional[List[Dict]]	Optional conditions that must be met.

2.7.6 app.models.automation.CreateAutomation

> **Source code in** `app/models/automation.py` ⌄

```python
30    class CreateAutomation(BaseModel):
31        """
32        Model for automation creation input.
33
34        Attributes:
35            name (str): Name of the automation.
36            description (str): Description of what the automation does.
37            user_id (str): ID of the user who owns this automation.
38            device_id (str): ID of the device associated with this automation.
39            enabled (bool): Whether the automation is enabled.
40            trigger_type (TriggerType): Type of trigger for this automation.
41            trigger_data (Dict): Configuration data for the trigger.
42            action_type (ActionType): Type of action for this automation.
43            action_data (Dict): Configuration data for the action.
44            conditions (Optional[List[Dict]]): Optional conditions that must be met.
45        """
46        name: str
47        description: str
48        user_id: str
49        device_id: str
50        enabled: bool = True
51        trigger_type: TriggerType
52        trigger_data: Dict[str, Any]
53        action_type: ActionType
54        action_data: Dict[str, Any]
55        conditions: Optional[List[Dict[str, Any]]] = None
56
57        @field_validator("name")
58        @classmethod
59        def validate_name(cls, v: str) -> str:
60            """
61            Validate automation name.
62
63            Arguments:
64                v (str): Name to be validated.
65
66            Returns:
67                str: Validated name.
68
69            Raises:
70                ValueError: Validation encountered a missing requirement.
71            """
72            if len(v) < 3:
73                raise ValueError("Automation name must be at least 3 characters long.")
74            if len(v) > 50:
75                raise ValueError("Automation name must be less than 50 characters long.")
76
77            return v
78
79        @field_validator("description")
80        @classmethod
81        def validate_description(cls, v: str) -> str:
82            """
83            Validate automation description.
84
85            Arguments:
86                v (str): Description to be validated.
87
88            Returns:
89                str: Validated description.
90
91            Raises:
92                ValueError: Validation encountered a missing requirement.
93            """
94            if len(v) > 500:
95                raise ValueError("Automation description must be less than 500 characters long.")
96
97            return v
```

`app.models.automation.CreateAutomation.validate_description(v)` `classmethod`

Validate automation description.

Parameters:

Name	Type	Description	Default
v	str	Description to be validated.	*required*

- 31/42 -

Returns:

Name	Type	Description
str	str	Validated description.

Raises:

Type	Description
ValueError	Validation encountered a missing requirement.

Source code in `app/models/automation.py`

```
79    @field_validator("description")
80    @classmethod
81    def validate_description(cls, v: str) -> str:
82        """
83        Validate automation description.
84
85        Arguments:
86            v (str): Description to be validated.
87
88        Returns:
89            str: Validated description.
90
91        Raises:
92            ValueError: Validation encountered a missing requirement.
93        """
94        if len(v) > 500:
95            raise ValueError("Automation description must be less than 500 characters long.")
96
97        return v
```

`app.models.automation.CreateAutomation.validate_name(v)` `classmethod`

Validate automation name.

Parameters:

Name	Type	Description	Default
v	str	Name to be validated.	*required*

Returns:

Name	Type	Description
str	str	Validated name.

Raises:

Type	Description
ValueError	Validation encountered a missing requirement.

Source code in `app/models/automation.py` ⌄

```
57    @field_validator("name")
58    @classmethod
59    def validate_name(cls, v: str) -> str:
60        """
61        Validate automation name.
62
63        Arguments:
64            v (str): Name to be validated.
65
66        Returns:
67            str: Validated name.
68
69        Raises:
70            ValueError: Validation encountered a missing requirement.
71        """
72        if len(v) < 3:
73            raise ValueError("Automation name must be at least 3 characters long.")
74        if len(v) > 50:
75            raise ValueError("Automation name must be less than 50 characters long.")
76
77        return v
```

2.7.7 `app.models.automation.TriggerType`

Bases: `str` , `Enum`

Enum for different types of automation triggers.

Source code in `app/models/automation.py` ⌄

```
10    class TriggerType(str, Enum):
11        """
12        Enum for different types of automation triggers.
13        """
14        TIME = "time"
15        SENSOR = "sensor"
16        MANUAL = "manual"
17        DEVICE_STATE = "device_state"
18        LOCATION = "location"
19        WEATHER = "weather"
```

2.8 `app.models.device`

Model for device validation & storage.

2.8.1 `app.models.device.CreateDevice`

Bases: `BaseModel`

Model for device registration input.

Attributes:

Name	Type	Description
name	str	Device name.
type	DeviceType	Type of device.
user_id	str	ID of the user who owns the device.
room_id	Optional[str]	ID of the room where the device is located.
ip_address	Optional[str]	Device IP address.
mac_address	Optional[str]	Device MAC address.
manufacturer	Optional[str]	Device manufacturer.
model	Optional[str]	Device model.
firmware_version	Optional[str]	Current firmware version.
settings	Optional[Dict]	Device-specific settings.

Source code in `app/models/device.py` ⌄

```python
36    class CreateDevice(BaseModel):
37        """
38        Model for device registration input.
39
40        Attributes:
41            name (str): Device name.
42            type (DeviceType): Type of device.
43            user_id (str): ID of the user who owns the device.
44            room_id (Optional[str]): ID of the room where the device is located.
45            ip_address (Optional[str]): Device IP address.
46            mac_address (Optional[str]): Device MAC address.
47            manufacturer (Optional[str]): Device manufacturer.
48            model (Optional[str]): Device model.
49            firmware_version (Optional[str]): Current firmware version.
50            settings (Optional[Dict]): Device-specific settings.
51        """
52        name: str
53        type: DeviceType
54        user_id: str
55        room_id: Optional[str] = None
56        ip_address: Optional[str] = None
57        mac_address: Optional[str] = None
58        manufacturer: Optional[str] = None
59        model: Optional[str] = None
60        firmware_version: Optional[str] = None
61        settings: Optional[Dict[str, Any]] = None
62
63        @field_validator("name")
64        @classmethod
65        def validate_name(cls, n: str) -> str:
66            """
67            Validate device name according to requirements.
68
69            Args:
70                name (str): Device name to be validated.
71
72            Returns:
73                str: Validated device name.
74
75            Raises:
76                ValueError: Validation encountered a missing requirement.
77            """
78            if len(n) < 1:
79                raise ValueError("Device name can't be empty.")
80            if len(n) > 100:
81                raise ValueError("Device name must be less than 100 characters long.")
82
83            return n
```

app.models.device.CreateDevice.validate_name(n) `classmethod`

Validate device name according to requirements.

Parameters:

Name	Type	Description	Default
name	str	Device name to be validated.	*required*

Returns:

Name	Type	Description
str	str	Validated device name.

Raises:

Type	Description
ValueError	Validation encountered a missing requirement.

Source code in `app/models/device.py` ⌄

```
63    @field_validator("name")
64    @classmethod
65    def validate_name(cls, n: str) -> str:
66        """
67        Validate device name according to requirements.
68
69        Args:
70            name (str): Device name to be validated.
71
72        Returns:
73            str: Validated device name.
74
75        Raises:
76            ValueError: Validation encountered a missing requirement.
77        """
78        if len(n) < 1:
79            raise ValueError("Device name can't be empty.")
80        if len(n) > 100:
81            raise ValueError("Device name must be less than 100 characters long.")
82
83        return n
```

2.8.2 `app.models.device.DeviceDB`

Bases: `BaseModel`

Internal model representing device data in the database

Attributes:

Name	Type	Description
id	str	Unique device identifier.
name	str	Device name.
type	DeviceType	Type of device.
user_id	str	ID of the user who owns the device.
room_id	Optional[str]	ID of the room where the device is located.
ip_address	Optional[str]	Device IP address.
mac_address	Optional[str]	Device MAC address.
manufacturer	Optional[str]	Device manufacturer.
model	Optional[str]	Device model.
firmware_version	Optional[str]	Current firmware version.
settings	Optional[Dict]	Device-specific settings.
status	DeviceStatus	Current device status.
last_online	Optional[datetime]	When the device was last seen online.
created	datetime	When the device was added to the system.
updated	Optional[datetime]	When the device data was last updated.
capabilities	List[str]	List of device capabilities/features.

Source code in `app/models/device.py` ⌄

```
86    class DeviceDB(BaseModel):
87        """
88        Internal model representing device data in the database
89
90        Attributes:
91            id (str): Unique device identifier.
92            name (str): Device name.
93            type (DeviceType): Type of device.
94            user_id (str): ID of the user who owns the device.
95            room_id (Optional[str]): ID of the room where the device is located.
96            ip_address (Optional[str]): Device IP address.
97            mac_address (Optional[str]): Device MAC address.
98            manufacturer (Optional[str]): Device manufacturer.
99            model (Optional[str]): Device model.
100           firmware_version (Optional[str]): Current firmware version.
101           settings (Optional[Dict]): Device-specific settings.
102           status (DeviceStatus): Current device status.
103           last_online (Optional[datetime]): When the device was last seen online.
104           created (datetime): When the device was added to the system.
105           updated (Optional[datetime]): When the device data was last updated.
106           capabilities (List[str]): List of device capabilities/features.
107       """
108       id: str
109       name: str
110       type: DeviceType
111       user_id: str
112       room_id: Optional[str] = None
113       ip_address: Optional[str] = None
114       mac_address: Optional[str] = None
115       manufacturer: Optional[str] = None
116       model: Optional[str] = None
117       firmware_version: Optional[str] = None
118       settings: Optional[Dict[str, Any]] = None
119       status: DeviceStatus = DeviceStatus.OFFLINE
120       last_online: Optional[datetime] = None
121       created: datetime = Field(default_factory=datetime.utcnow)
122       updated: Optional[datetime] = None
123       capabilities: List[str] = []
124
125       model_config = ConfigDict(from_attributes=True)
```

2.8.3 `app.models.device.DeviceResponse`

Bases: `BaseModel`

Model for device data returned in API responses.

Attributes:

Name	Type	Description
id	str	Unique device identifier.
name	str	Device name.
type	DeviceType	Type of device.
user_id	str	ID of the user who owns the device.
room_id	Optional[str]	ID of the room where the device is located.
manufacturer	Optional[str]	Device manufacturer.
model	Optional[str]	Device model.
status	DeviceStatus	Current device status.
last_online	Optional[datetime]	When the device was last seen online.
created	datetime	When the device was added to the system.
capabilities	List[str]	List of device capabilities/features.

Source code in `app/models/device.py`

```python
128    class DeviceResponse(BaseModel):
129        """
130        Model for device data returned in API responses.
131
132        Attributes:
133            id (str): Unique device identifier.
134            name (str): Device name.
135            type (DeviceType): Type of device.
136            user_id (str): ID of the user who owns the device.
137            room_id (Optional[str]): ID of the room where the device is located.
138            manufacturer (Optional[str]): Device manufacturer.
139            model (Optional[str]): Device model.
140            status (DeviceStatus): Current device status.
141            last_online (Optional[datetime]): When the device was last seen online.
142            created (datetime): When the device was added to the system.
143            capabilities (List[str]): List of device capabilities/features.
144        """
145        id: str
146        name: str
147        type: DeviceType
148        user_id: str
149        room_id: Optional[str] = None
150        manufacturer: Optional[str] = None
151        model: Optional[str] = None
152        status: DeviceStatus
153        last_online: Optional[datetime] = None
154        created: datetime
155        capabilities: List[str] = []
```

2.8.4 `app.models.device.DeviceStatus`

Bases: `str`, `Enum`

Enumeration of possible device statuses.

> **Source code in** `app/models/device.py` ∨
>
> ```
> 26    class DeviceStatus(str, Enum):
> 27        """
> 28        Enumeration of possible device statuses.
> 29        """
> 30        ONLINE = "online"
> 31        OFFLINE = "offline"
> 32        ERROR = "error"
> 33        MAINTENANCE = "maintenance"
> ```

2.8.5 `app.models.device.DeviceType`

Bases: `str`, `Enum`

Enumeration of supported device types.

> **Source code in** `app/models/device.py` ∨
>
> ```
> 11    class DeviceType(str, Enum):
> 12        """
> 13        Enumeration of supported device types.
> 14        """
> 15        LIGHT = "light"
> 16        THERMOSTAT = "thermostat"
> 17        LOCK = "lock"
> 18        CAMERA = "camera"
> 19        SENSOR = "sensor"
> 20        SWITCH = "switch"
> 21        OUTLET = "outlet"
> 22        SPEAKER = "speaker"
> 23        OTHER = "other"
> ```

2.8.6 `app.models.device.DeviceUpdate`

Bases: `BaseModel`

Model for updating device information.

Attributes:

Name	Type	Description
name	Optional[str]	Updated device name.
room_id	Optional[str]	Updated room location.
ip_address	Optional[str]	Updated IP address.
firmware_version	Optional[str]	Updated firmware version.
settings	Optional[Dict]	Updated device settings.
status	Optional[DeviceStatus]	Updated device status.

Source code in `app/models/device.py` ⌄

```
158   class DeviceUpdate(BaseModel):
159       """
160       Model for updating device information.
161
162       Attributes:
163           name (Optional[str]): Updated device name.
164           room_id (Optional[str]): Updated room location.
165           ip_address (Optional[str]): Updated IP address.
166           firmware_version (Optional[str]): Updated firmware version.
167           settings (Optional[Dict]): Updated device settings.
168           status (Optional[DeviceStatus]): Updated device status.
169       """
170       name: Optional[str] = None
171       room_id: Optional[str] = None
172       ip_address: Optional[str] = None
173       firmware_version: Optional[str] = None
174       settings: Optional[Dict[str, Any]] = None
175       status: Optional[DeviceStatus] = None
176
177       @field_validator("name")
178       @classmethod
179       def validate_name(cls, n: Optional[str]) -> Optional[str]:
180           """
181           Validate device name according to requirements.
182
183           Args:
184               n (Optional[str]): Device name to be validated.
185
186           Returns:
187               Optional[str]: Validated device name.
188
189           Raises:
190               ValueError: Validation encountered a missing requirement.
191           """
192           if n is None:
193               return None
194
195           if len(n) < 1:
196               raise ValueError("Device name can't be empty.")
197           if len(n) > 100:
198               raise ValueError("Device name must be less than 100 characters long.")
199
200           return n
```

`app.models.device.DeviceUpdate.validate_name(n)` `classmethod`

Validate device name according to requirements.

Parameters:

Name	Type	Description	Default
n	Optional[str]	Device name to be validated.	*required*

Returns:

Type	Description
Optional[str]	Optional[str]: Validated device name.

Raises:

Type	Description
ValueError	Validation encountered a missing requirement.

> **Source code in** `app/models/device.py` ∨

```
177    @field_validator("name")
178    @classmethod
179    def validate_name(cls, n: Optional[str]) -> Optional[str]:
180        """
181        Validate device name according to requirements.
182
183        Args:
184            n (Optional[str]): Device name to be validated.
185
186        Returns:
187            Optional[str]: Validated device name.
188
189        Raises:
190            ValueError: Validation encountered a missing requirement.
191        """
192        if n is None:
193            return None
194
195        if len(n) < 1:
196            raise ValueError("Device name can't be empty.")
197        if len(n) > 100:
198            raise ValueError("Device name must be less than 100 characters long.")
199
200        return n
```

2.9 `app.models.goal`

Models for energy goal validation and storage.

2.9.1 `app.models.goal.CreateEnergyGoal`

Bases: `BaseModel`

Model for energy goal creation input.

Attributes:

Name	Type	Description
`user_id`	`str`	ID of the user who owns the goal.
`title`	`str`	Title of the goal.
`description`	`str`	Detailed description of the goal.
`type`	`GoalType`	Type of energy goal.
`target_value`	`float`	Target value for the goal (e.g., kWh to save).
`timeframe`	`GoalTimeframe`	Timeframe for the goal.
`start_date`	`datetime`	When the goal starts.
`end_date`	`Optional[datetime]`	When the goal ends (required for CUSTOM timeframe).
`related_devices`	`Optional[List[str]]`	List of device IDs this goal applies to.

Source code in `app/models/goal.py` ˅

Source code in `app/models/goal.py` ˅

```
 43
 44
 45
 46
 47
 48
 49
 50
 51
 52
 53
 54
 55
 56
 57
 58
 59
 60
 61
 62
 63
 64
 65
 66
 67
 68
 69
 70
 71
 72
 73
 74
 75
 76
 77
 78
 79
 80
 81
 82
 83
 84
 85
 86
 87
 88
 89
 90
 91
 92
 93
 94
 95
 96
 97
 98
 99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
```