Sync Smart Home API Docs

None

None

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`

```python
@app.get("/")
async def root():
    """
    API health check endpoint.
    """
    return {
        "status": "online",
        "message": "Sync Smart Home API is running",
        "version": "1.0.0"
    }
```

2.1.2 `app.main.startup_event()`

Initialize database on startup.

Source code in `app/main.py`

```python
@app.on_event("startup")
def startup_event():
    """Initialize database on startup."""
    init_db()

    # Create reports directory if it doesn't exist
    import os
    from app.utils.report.report_generator import REPORTS_DIR
    os.makedirs(REPORTS_DIR, exist_ok=True)
```

2.2 `app.generate_report`

CLI script to generate energy reports directly from the command line.

2.2.1 `app.generate_report.get_user_devices(user_id)`

Get all devices for a user.

Source code in `app/generate_report.py`

```python
def get_user_devices(user_id):
    """Get all devices for a user."""
    devices = list(d_c.find({"user_id": user_id}))
    return devices
```

2.2.2 `app.generate_report.main()`

Generate a report based on command line arguments.

Source code in `app/generate_report.py`

```python
def main():
    """Generate a report based on command line arguments."""
    args = parse_args()

    # Validate user
    user = validate_user_id(args.user_id)
    print(f"Generating report for user: {user.get('username', user.get('email', 'Unknown'))}")

    # Determine date range
    if args.historical:
        # Use March 2024 dates where we know data exists
        end_date = datetime(2024, 3, 19)  # Latest record date
        start_date = end_date - timedelta(days=args.days)
        print("Using historical data period (March 2024)")
    else:
        # Use current dates
        end_date = datetime.now()
        start_date = end_date - timedelta(days=args.days)
        print("Using current date period")

    start_date_str = start_date.strftime("%Y-%m-%d")
    end_date_str = end_date.strftime("%Y-%m-%d")
    print(f"Date range: {start_date_str} to {end_date_str}")

    # Get device IDs if not specified
    device_ids = args.device_ids
    if not device_ids:
        devices = get_user_devices(args.user_id)
        device_ids = [device["id"] for device in devices]
        print(f"Using all devices ({len(device_ids)}) for the user")
    else:
        print(f"Using specified devices: {', '.join(device_ids)}")

    # Create a title if not specified
    title = args.title
    if not title:
        title = f"Energy Report ({start_date_str} to {end_date_str})"

    # Generate a UUID for the report
    report_uuid = str(uuid.uuid4())

    # Create report object
    report_db = ReportDB(
        id=report_uuid,
        user_id=args.user_id,
        title=title,
        format=args.format.lower(),
        report_type=args.report_type,
        start_date=start_date_str,
        end_date=end_date_str,
        device_ids=device_ids,
        status="pending"
    )

    # Create report in database
    print("Creating report record...")
    mongo_id = ReportService.create_report(report_db)
    print(f"Report record created with MongoDB ID: {mongo_id}")
    print(f"Report UUID: {report_uuid}")

    # Generate the report using the UUID we assigned
    print("Generating report...")
    success, file_path, error = ReportService.generate_report(report_uuid)

    if success and file_path:
        print(f"Report successfully generated!")
        print(f"Report file: {file_path}")
    else:
        print(f"Error generating report: {error}")
```

2.2.3 `app.generate_report.parse_args()`

Parse command line arguments.

Source code in `app/generate_report.py`

```python
def parse_args():
    """Parse command line arguments."""
    parser = argparse.ArgumentParser(description="Generate energy usage reports")

    parser.add_argument("--user_id", required=True, help="User ID to generate report for")
    parser.add_argument("--format", choices=["pdf", "csv"], default="pdf", help="Report format")
    parser.add_argument("--days", type=int, default=30, help="Number of days to include in report")
    parser.add_argument("--title", help="Report title")
    parser.add_argument("--device_ids", nargs="+", help="Specific device IDs to include (space-separated)")
    parser.add_argument("--report_type", default="energy", help="Report type (default: energy)")
    parser.add_argument("--historical", action="store_true", help="Use historical data (March 2024)")

    return parser.parse_args()
```

2.2.4 `app.generate_report.validate_user_id(user_id)`

Check if the user exists in the database.

Source code in `app/generate_report.py`

```python
def validate_user_id(user_id):
    """Check if the user exists in the database."""
    user = u_c.find_one({"id": user_id})
    if not user:
        print(f"Error: User with ID {user_id} not found.")
        sys.exit(1)
    return user
```

2.3 `app.data_util`

Utility to check for usage data and optionally generate test data.

2.3.1 `app.data_util.check_usage_data(user_id, days=30)`

Check if usage data exists for a user's devices.

Source code in `app/data_util.py`

```python
def check_usage_data(user_id, days=30):
    """Check if usage data exists for a user's devices."""
    # Get devices for the user
    devices = list(d_c.find({"user_id": user_id}))
    device_ids = [device["id"] for device in devices]

    if not device_ids:
        print(f"No devices found for user {user_id}")
        return False

    print(f"Found {len(device_ids)} devices for user {user_id}")

    # Check for usage data within the specified time range
    end_date = datetime.now()
    start_date = end_date - timedelta(days=days)

    query = {
        "device_id": {"$in": device_ids},
        "timestamp": {
            "$gte": start_date,
            "$lte": end_date
        }
    }

    count = us_c.count_documents(query)

    if count > 0:
        print(f"Found {count} usage records in the last {days} days")
        return True
    else:
        print(f"No usage records found in the last {days} days")
        # Check if there's any usage data at all for these devices
        all_time_count = us_c.count_documents({"device_id": {"$in": device_ids}})
        if all_time_count > 0:
            print(f"Found {all_time_count} usage records for these devices across all time")
            # Show the earliest and latest records
            earliest = us_c.find({"device_id": {"$in": device_ids}}).sort("timestamp", 1).limit(1)
            latest = us_c.find({"device_id": {"$in": device_ids}}).sort("timestamp", -1).limit(1)

            earliest_list = list(earliest)
            latest_list = list(latest)

            if earliest_list:
                print(f"Earliest record: {earliest_list[0].get('timestamp')}")
            if latest_list:
                print(f"Latest record: {latest_list[0].get('timestamp')}")
        else:
            print("No usage records found for these devices at all")
        return False
```

2.3.2 `app.data_util.generate_test_data(user_id, days=30, records_per_device=5)`

Generate test usage data for a user's devices.

Source code in `app/data_util.py`

```python
def generate_test_data(user_id, days=30, records_per_device=5):
    """Generate test usage data for a user's devices."""
    # Get devices for the user
    devices = list(d_c.find({"user_id": user_id}))

    if not devices:
        print(f"No devices found for user {user_id}")
        return False

    print(f"Generating test data for {len(devices)} devices")

    # Generate data
    end_date = datetime.now()
    start_date = end_date - timedelta(days=days)

    generated_count = 0

    for device in devices:
        device_id = device["id"]
        device_name = device.get("name", "Unknown Device")

        for _ in range(records_per_device):
            # Generate a random timestamp in the date range
            random_days = random.randint(0, days)
            timestamp = end_date - timedelta(days=random_days)

            # Generate random energy consumption (0.1 to 5.0 kWh)
            energy_consumed = round(random.uniform(0.1, 5.0), 2)

            # Generate random duration (10 to 480 minutes)
            duration = random.randint(10, 480)

            # Create usage record
            usage_record = {
                "id": str(uuid.uuid4()),
                "device_id": device_id,
                "timestamp": timestamp,
                "energy_consumed": energy_consumed,
                "duration": duration,
                "status": random.choice(["active", "idle"]),
                "metrics": {
                    "temperature": round(random.uniform(18, 28), 1) if random.random() > 0.5 else None,
                    "brightness": random.randint(10, 100) if "light" in device_name.lower() else None,
                    "usage_minutes": duration
                },
                "created": timestamp
            }

            # Insert into the database
            us_c.insert_one(usage_record)
            generated_count += 1

    print(f"Generated {generated_count} test usage records")
    return True
```

2.4 `app.inspect_data`

Script to inspect and fix usage data for generating reports.

2.4.1 `app.inspect_data.inspect_usage_data(user_id, start_date, end_date)`

Inspect usage data for a user in a specific date range.

Source code in `app/inspect_data.py`

```python
def inspect_usage_data(user_id, start_date, end_date):
    """Inspect usage data for a user in a specific date range."""
    # Get devices for the user
    devices = list(d_c.find({"user_id": user_id}))
    device_ids = [device["id"] for device in devices]

    if not device_ids:
        print(f"No devices found for user {user_id}")
        return

    # Build query
    query = {
        "device_id": {"$in": device_ids},
        "timestamp": {
            "$gte": start_date,
            "$lte": end_date
        }
    }

    # Fetch the records
    records = list(us_c.find(query))
    print(f"Found {len(records)} records in the date range")

    # Check for missing fields
    fields_to_check = ["energy_consumed", "duration", "status", "metrics"]
    missing_data = {}

    for field in fields_to_check:
        missing_data[field] = 0

    for record in records:
        for field in fields_to_check:
            if field not in record or record[field] is None:
                missing_data[field] += 1

    print("\nMissing fields summary:")
    for field, count in missing_data.items():
        if count > 0:
            print(f"- {field}: Missing in {count} records ({count/len(records)*100:.1f}%)")

    # Show a few example records
    if records:
        print("\nSample record structure:")
        for key, value in records[0].items():
            print(f"- {key}: {type(value).__name__} = {value}")

    # Check for None values in energy_consumed
    none_energy = [r for r in records if "energy_consumed" not in r or r["energy_consumed"] is None]
    if none_energy:
        print(f"\nFound {len(none_energy)} records with None energy_consumed values")
        print("Would you like to fix these records? (y/n)")
        response = input().strip().lower()

        if response == 'y':
            for record in none_energy:
                # Update with default value
                us_c.update_one(
                    {"_id": record["_id"]},
                    {"$set": {"energy_consumed": 0.0}}
                )
            print(f"Fixed {len(none_energy)} records")

    # Return the records for additional processing
    return records
```

2.5 `app.patched_report_generator`

Patched version of report generator that handles None values.

2.5.1 `app.patched_report_generator.fetch_energy_data(user_id, start_date, end_date, device_ids=None)`

Patched version of fetch_energy_data that handles None values.

Source code in `app/patched_report_generator.py`

```python
def fetch_energy_data(user_id, start_date, end_date, device_ids=None):
    """
    Patched version of fetch_energy_data that handles None values.
    """
    # Build the query
    query = {}

    # Filter by device IDs
    if device_ids:
        query["device_id"] = {"$in": device_ids}
    else:
        # Get all devices owned by the user
        user_devices = list(d_c.find({"user_id": user_id}))
        if not user_devices:
            return []

        user_device_ids = [device["id"] for device in user_devices]
        if user_device_ids:
            query["device_id"] = {"$in": user_device_ids}
        else:
            return []

    # Add date range filter
    if start_date or end_date:
        timestamp_query = {}
        if start_date:
            timestamp_query["$gte"] = start_date
        if end_date:
            timestamp_query["$lte"] = end_date

        if timestamp_query:
            query["timestamp"] = timestamp_query

    # Execute the query
    print(f"Query: {query}")
    cursor = us_c.find(query).sort("timestamp", 1)
    print(f"Cursor type: {type(cursor)}")
    usage_data = list(cursor)
    print(f"Usage data type: {type(usage_data)}, length: {len(usage_data)}")

    # Fix missing or None values
    for i, record in enumerate(usage_data):
        # Fill in missing energy_consumed with zeros
        if "energy_consumed" not in record or record["energy_consumed"] is None:
            print(f"Fixing record {i}: Adding default energy_consumed = 0.0")
            record["energy_consumed"] = 0.0

        # Convert timestamp to ISO format if it's a datetime object
        if "timestamp" in record and isinstance(record["timestamp"], datetime):
            record["timestamp"] = record["timestamp"].isoformat()

    # Enhance usage data with device information
    enhanced_data = []
    for record in usage_data:
        # Get device info
        device_id = record.get("device_id")
        device = d_c.find_one({"id": device_id})

        # Create enhanced record with location
        enhanced_record = {
            "timestamp": record.get("timestamp"),
            "device_id": device_id,
            "energy_consumed": record.get("energy_consumed", 0.0),  # Default to 0 if missing
            "location": device.get("room_id") if device else "Unknown"
        }

        enhanced_data.append(enhanced_record)

    return enhanced_data
```

2.5.2 `app.patched_report_generator.fetch_user_data(user_id)`

Fetch user data for report personalization.

Source code in `app/patched_report_generator.py`

```python
def fetch_user_data(user_id):
    """Fetch user data for report personalization."""
    user = u_c.find_one({"id": user_id})

    if not user:
        return {}

    return {
        "email": user.get("email"),
        "username": user.get("username")
    }
```

2.5.3 `app.patched_report_generator.generate_energy_report(energy_data, user_data=None, format='pdf', start_date=None, end_date=None)`

Patched version that handles the path to the report generator.

Source code in `app/patched_report_generator.py`

```python
def generate_energy_report(energy_data, user_data=None, format="pdf", start_date=None, end_date=None):
    """
    Patched version that handles the path to the report generator.
    """
    from app.utils.report.report_generator import generate_energy_report as gen_report

    # Print some debug info
    print(f"Generating {format} report with {len(energy_data)} records")
    if len(energy_data) > 0:
        sample = energy_data[0]
        print(f"Sample record: {sample}")

    # Call the original function
    try:
        report_path = gen_report(
            energy_data=energy_data,
            user_data=user_data,
            format=format.lower(),
            start_date=start_date,
            end_date=end_date
        )
        return report_path
    except Exception as e:
        traceback.print_exc()
        print(f"Error in report generation: {e}")
        return None
```

2.5.4 `app.patched_report_generator.generate_report(report_id)`

Patched version of generate_report that uses our fixed functions.

Source code in `app/patched_report_generator.py`

```python
def generate_report(report_id):
    """
    Patched version of generate_report that uses our fixed functions.
    """
    # Get report data
    report_data = r_c.find_one({"id": report_id})
    if not report_data:
        return False, None, "Report not found"

    try:
        # Update status to generating
        r_c.update_one(
            {"id": report_id},
            {"$set": {"status": "generating", "updated": datetime.utcnow()}}
        )

        # Fetch energy data with our patched function
        start_date = None
        end_date = None

        if report_data.get("start_date"):
            start_date = datetime.strptime(report_data["start_date"], "%Y-%m-%d")
        if report_data.get("end_date"):
            end_date = datetime.strptime(report_data["end_date"], "%Y-%m-%d") + timedelta(days=1) - timedelta(seconds=1)

        energy_data = fetch_energy_data(
            user_id=report_data["user_id"],
            start_date=start_date,
            end_date=end_date,
            device_ids=report_data.get("device_ids")
        )

        if not energy_data:
            error_msg = "No energy data found for the specified criteria"
            r_c.update_one(
                {"id": report_id},
                {"$set": {"status": "failed", "error_message": error_msg, "updated": datetime.utcnow()}}
            )
            return False, None, error_msg

        # Fetch user data for personalization
        user_data = fetch_user_data(report_data["user_id"])

        # Generate the report with our patched function
        report_path = generate_energy_report(
            energy_data=energy_data,
            user_data=user_data,
            format=report_data["format"].lower(),
            start_date=report_data.get("start_date"),
            end_date=report_data.get("end_date")
        )

        if not report_path:
            error_msg = "Failed to generate report file"
            r_c.update_one(
                {"id": report_id},
                {"$set": {"status": "failed", "error_message": error_msg, "updated": datetime.utcnow()}}
            )
            return False, None, error_msg

        # Update the report record with success status
        r_c.update_one(
            {"id": report_id},
            {"$set": {
                "status": "completed",
                "file_path": report_path,
                "completed": datetime.utcnow(),
                "updated": datetime.utcnow()
            }}
        )

        return True, report_path, None

    except Exception as e:
        # Update the report record with failure status
        error_message = str(e)
        r_c.update_one(
            {"id": report_id},
            {"$set": {"status": "failed", "error_message": error_message, "updated": datetime.utcnow()}}
        )

        return False, None, error_message
```

2.5.5 `app.patched_report_generator.get_user_devices(user_id)`

Get all devices for a user.

Source code in `app/patched_report_generator.py`

```python
def get_user_devices(user_id):
    """Get all devices for a user."""
    devices = list(d_c.find({"user_id": user_id}))
    return devices
```

2.5.6 `app.patched_report_generator.main()`

Patched report generation script.

Source code in `app/patched_report_generator.py`

```python
def main():
    """Patched report generation script."""
    parser = argparse.ArgumentParser(description="Generate energy usage reports")

    parser.add_argument("--user_id", required=True, help="User ID to generate report for")
    parser.add_argument("--format", choices=["pdf", "csv"], default="pdf", help="Report format")
    parser.add_argument("--days", type=int, default=30, help="Number of days to include in report")
    parser.add_argument("--title", help="Report title")
    parser.add_argument("--device_ids", nargs="+", help="Specific device IDs to include (space-separated)")
    parser.add_argument("--report_type", default="energy", help="Report type (default: energy)")
    parser.add_argument("--historical", action="store_true", help="Use historical data (March 2024)")

    args = parse_args = parser.parse_args()

    # Validate user
    user = validate_user_id(args.user_id)
    print(f"Generating report for user: {user.get('username', user.get('email', 'Unknown'))}")

    # Determine date range
    if args.historical:
        # Use March 2024 dates where we know data exists
        end_date = datetime(2024, 3, 19)  # Latest record date
        start_date = end_date - timedelta(days=args.days)
        print("Using historical data period (March 2024)")
    else:
        # Use current dates
        end_date = datetime.now()
        start_date = end_date - timedelta(days=args.days)
        print("Using current date period")

    start_date_str = start_date.strftime("%Y-%m-%d")
    end_date_str = end_date.strftime("%Y-%m-%d")
    print(f"Date range: {start_date_str} to {end_date_str}")

    # Get device IDs if not specified
    device_ids = args.device_ids
    if not device_ids:
        devices = get_user_devices(args.user_id)
        device_ids = [device["id"] for device in devices]
        print(f"Using all devices ({len(device_ids)}) for the user")
    else:
        print(f"Using specified devices: {', '.join(device_ids)}")

    # Create a title if not specified
    title = args.title
    if not title:
        title = f"Energy Report ({start_date_str} to {end_date_str})"

    # Generate a UUID for the report
    report_uuid = str(uuid.uuid4())

    # Create report object
    report_db = ReportDB(
        id=report_uuid,
        user_id=args.user_id,
        title=title,
        format=args.format.lower(),
        report_type=args.report_type,
        start_date=start_date_str,
        end_date=end_date_str,
        device_ids=device_ids,
        status="pending"
    )

    # Create report in database
    print("Creating report record...")
    result = r_c.insert_one(report_db.model_dump())
    print(f"Report record created with MongoDB ID: {result.inserted_id}")
    print(f"Report UUID: {report_uuid}")

    # Generate the report using the UUID we assigned
    print("Generating report...")
    success, file_path, error = generate_report(report_uuid)

    if success and file_path:
        print(f"Report successfully generated!")
        print(f"Report file: {file_path}")
    else:
        print(f"Error generating report: {error}")
```

2.5.7 `app.patched_report_generator.validate_user_id(user_id)`

Check if the user exists in the database.

Source code in `app/patched_report_generator.py`

```python
def validate_user_id(user_id):
    """Check if the user exists in the database."""
    user = u_c.find_one({"id": user_id})
    if not user:
        print(f"Error: User with ID {user_id} not found.")
        sys.exit(1)
    return user
```

2.6 `app.core.auth`

Authentication dependencies for FastAPI.

2.6.1 `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
async def get_current_user(credentials: HTTPBasicCredentials = Depends(security)) -> UserDB:
    """
    Get the currently authenticated user.

    Uses HTTP Basic Authentication to validate the user.

    Args:
        credentials: The HTTP Basic credentials from the request

    Returns:
        UserDB: The authenticated user

    Raises:
        HTTPException: If authentication fails
    """
    auth_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Invalid username or password",
        headers={"WWW-Authenticate": "Basic"},
    )

    # Try to find user by username
    user = u_c.find_one({"username": credentials.username})

    # If not found, try by email
    if not user:
        user = u_c.find_one({"email": credentials.username})

    # Check credentials
    if not user or not verify_password(credentials.password, user["hashed_password"]):
        raise auth_exception

    # Check if user is active
    if not user.get("active", True):  # Default to active if not specified
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Inactive user account",
            headers={"WWW-Authenticate": "Basic"},
        )

    return UserDB(**user)
```

2.7 `app.core.password`

Implements password security features.

2.7.1 `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`

```python
def hash_password(p: str) -> str:
    """
    Hashes a password.

    Args:
        p (str): Password to hash.

    Returns:
        str: Hashed password.

    Raises:
        ValueError: If password fails to hash.
    """
    try:
        return pc.hash(p)
    except Exception as e:
        raise ValueError(f"Error hashing password: {e}") from e
```

2.7.2 `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`

```python
def verify_password(p: str, h: str) -> bool:
    """
    Matches plain-text password with the hashed password.

    Args:
        p (str): Plain-text password.
        h (str): Hashed password.

    Returns:
        bool: (True: Passwords match); (False: Passwords don't match).

    Raises:
        ValueError: If passwords fail to compare.
    """
    try:
        return pc.verify(p, h)
    except Exception as e:
        raise ValueError(f"Error verifying password: {e}") from e
```

2.7.3 `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`

```python
def verify_role(u: str, r: str):
    """
    Enforces correct user role for access.

    Args:
        u (str): User's current role.
        r (str): Required user role.

    Raises:
        HTTPException: If user lacks the required role.
    """
    if u != r:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail=f"Permission denied. {r} role is required for access."
        )

    return True
```

2.8 `app.db.data`

Handles MongoDB database connections & operations.

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

Initialize MongoDB database & creates indexes.

Source code in `app/db/data.py`

```python
def init_db():
    """
    Initialize MongoDB database & creates indexes.
    """
    # User collection
    u_c.create_index("id", unique=True)     # Unique identification
    u_c.create_index("email", unique=True)  # Unique email address
    u_c.create_index("username")            # General username

    # Profile collection
    p_c.create_index("id", unique=True)         # Unique identification
    p_c.create_index("user_id", unique=True)    # Unique user identification

    # Device collection
    d_c.create_index("id", unique=True)             # Device identification
    d_c.create_index("user_id")                     # User identification
    d_c.create_index([("type", 1), ("user_id", 1)]) # Filter type through user identification

    # Room collection
    r_c.create_index("id", unique=True)     # Unique identification
    r_c.create_index("user_id")             # User identification
    r_c.create_index("home_id")             # Home identification

    # Usage collection
    us_c.create_index("id", unique=True)                     # Unique identification
    us_c.create_index("device_id")                           # Device identification
    us_c.create_index("timestamp")                           # Usage log
    us_c.create_index([("device_id", 1), ("timestamp", -1)])  # Filter log by device identification

    # Automation collection
    a_c.create_index("id", unique=True)     # Unique identification
    a_c.create_index("user_id")             # User identification
    a_c.create_index("device_id")           # User identification

    # Notification collection
    n_c.create_index("id", unique=True)                                 # Unique identification
    n_c.create_index("user_id")                                         # User identification
    n_c.create_index([("user_id", 1), ("read", 1), ("timestamp", -1)])  # Filter notification read by device & time

    # Access Management collection
    am_c.create_index("id", unique=True)                        # Unique identification
    am_c.create_index("owner_id")                               # Owner identification
    am_c.create_index("resource_id")                            # Resource identification
    am_c.create_index([("owner_id", 1), ("resource_id", 1)])    # Filters resource by its owner

    # Goal collection
    g_c.create_index("id", unique=True)                 # Unique identification
    g_c.create_index("user_id")                         # User identification
    g_c.create_index("type")                            # Type of goal
    g_c.create_index([("user_id", 1), ("type", 1)])     # Filters types of goals by user identification

    # Analytics collection
    an_c.create_index("id", unique=True)                    # Unique identification
    an_c.create_index("user_id")                            # User identification
    an_c.create_index("device_id")                          # Device identification
    an_c.create_index([("user_id", 1), ("timestamp", -1)])  # Filters user identification by timestamp

    # Suggestion collection
    s_c.create_index("id", unique=True)                                     # Unique identification
    s_c.create_index("user_id")                                             # User identification
    s_c.create_index([("user_id", 1), ("status", 1), ("timestamp", -1)])    # Filters user identification with status by timestamp

    print("Database initialized with indexes.")
```

2.9 `app.models.access_management`

Models for access management validation.

2.9.1 `app.models.access_management.AccessLevel`

Bases: `str`, `Enum`

Enum for access permission levels.

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

```
class AccessLevel(str, Enum):
    """
    Enum for access permission levels.
    """
    READ = "read"
    CONTROL = "control"
    MANAGE = "manage"
    ADMIN = "admin"
```

2.9.2 `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`

```python
class AccessManagementDB(BaseModel):
    """
    Internal model representing access management data in the database.

    Attributes:
        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.
    """
    id: str = Field(default_factory=lambda: str(uuid.uuid4()))
    owner_id: str
    resource_id: str
    resource_type: ResourceType
    user_id: str
    access_level: AccessLevel
    created: datetime = Field(default_factory=datetime.utcnow)
    updated: Optional[datetime] = None
    expires_at: Optional[datetime] = None
    active: bool = True
    note: Optional[str] = None

    model_config = ConfigDict(from_attributes=True)
```

2.9.3 `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`

```python
class AccessManagementResponse(BaseModel):
    """
    Model for access management data returned in API responses.

    Attributes:
        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.
    """
    id: str
    owner_id: str
    resource_id: str
    resource_type: ResourceType
    user_id: str
    access_level: AccessLevel
    created: datetime
    expires_at: Optional[datetime] = None
    active: bool
    note: Optional[str] = None

    model_config = ConfigDict(from_attributes=True)
```

2.9.4 `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`

```python
class AccessManagementUpdate(BaseModel):
    """
    Model for updating access management entries.

    Attributes:
        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.
    """
    access_level: Optional[AccessLevel] = None
    expires_at: Optional[datetime] = None
    active: Optional[bool] = None
    note: Optional[str] = None

    @field_validator("note")
    @classmethod
    def validate_note(cls, n: Optional[str]) -> Optional[str]:
        """
        Validate note length.

        Arguments:
            n (Optional[str]): Note to validate.

        Returns:
            Optional[str]: Validated note.

        Raises:
            ValueError: If validation fails.
        """
        if n is not None and len(n) > 200:
            raise ValueError("Note must be less than 200 characters long.")

        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`

```python
@field_validator("note")
@classmethod
def validate_note(cls, n: Optional[str]) -> Optional[str]:
    """
    Validate note length.

    Arguments:
        n (Optional[str]): Note to validate.

    Returns:
        Optional[str]: Validated note.

    Raises:
        ValueError: If validation fails.
    """
    if n is not None and len(n) > 200:
        raise ValueError("Note must be less than 200 characters long.")

    return n
```

2.9.5 `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`

```python
class CreateAccessManagement(BaseModel):
    """
    Model for creating access management entries.

    Attributes:
        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.
    """
    owner_id: str
    resource_id: str
    resource_type: ResourceType
    user_ids: List[str]
    access_level: AccessLevel
    expires_at: Optional[datetime] = None
    note: Optional[str] = None

    @field_validator("note")
    @classmethod
    def validate_note(cls, n: Optional[str]) -> Optional[str]:
        """
        Validate note length.

        Arguments:
            n (Optional[str]): Note to validate.

        Returns:
            Optional[str]: Validated note.

        Raises:
            ValueError: If validation fails.
        """
        if n is not None and len(n) > 200:
            raise ValueError("Note must be less than 200 characters long.")

        return n

    @field_validator("user_ids")
    @classmethod
    def validate_user_ids(cls, u: List[str]) -> List[str]:
        """
        Validate user IDs list.

        Arguments:
            u (List[str]): List of user IDs.

        Returns:
            List[str]: Validated list of user IDs.

        Raises:
            ValueError: If validation fails.
        """
        if not u:
            raise ValueError("At least one user ID must be provided.")
        if len(u) > 50:
            raise ValueError("Cannot share with more than 50 users at once.")

        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`

```
@field_validator("note")
@classmethod
def validate_note(cls, n: Optional[str]) -> Optional[str]:
    """
    Validate note length.

    Arguments:
        n (Optional[str]): Note to validate.

    Returns:
        Optional[str]: Validated note.

    Raises:
        ValueError: If validation fails.
    """
    if n is not None and len(n) > 200:
        raise ValueError("Note must be less than 200 characters long.")

    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`

```python
@field_validator("user_ids")
@classmethod
def validate_user_ids(cls, u: List[str]) -> List[str]:
    """
    Validate user IDs list.

    Arguments:
        u (List[str]): List of user IDs.

    Returns:
        List[str]: Validated list of user IDs.

    Raises:
        ValueError: If validation fails.
    """
    if not u:
        raise ValueError("At least one user ID must be provided.")
    if len(u) > 50:
        raise ValueError("Cannot share with more than 50 users at once.")

    return u
```

2.9.6 `app.models.access_management.ResourceType`

Bases: `str`, `Enum`

Enum for supported resource types in the system.

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

```python
class ResourceType(str, Enum):
    """
    Enum for supported resource types in the system.
    """
    DEVICE = "device"
    ROOM = "room"
    HOME = "home"
    AUTOMATION = "automation"
```

2.10 `app.models.analytics`

Models for analytics validation and storage.

2.10.1 `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`

```python
class AnalyticsDB(BaseModel):
    """
    Internal model representing analytics data in the database.

    Attributes:
        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.
    """
    id: str = Field(default_factory=lambda: str(uuid.uuid4()))
    user_id: str
    device_id: str
    data_type: str
    metrics: Dict[str, Any]
    tags: List[str] = []
    timestamp: datetime = Field(default_factory=datetime.utcnow)
    updated: Optional[datetime] = None

    model_config = ConfigDict(from_attributes=True)
```

2.10.2 `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`

```python
class AnalyticsQuery(BaseModel):
    """
    Model for querying analytics data with filters.

    Attributes:
        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.
    """
    user_id: Optional[str] = None
    device_id: Optional[str] = None
    data_type: Optional[str] = None
    start_time: Optional[datetime] = None
    end_time: Optional[datetime] = None
    tags: Optional[List[str]] = None

    @field_validator("start_time", "end_time")
    @classmethod
    def validate_time_range(cls, v: Optional[datetime], info) -> Optional[datetime]:
        """
        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.
        """
        return v

    def validate_time_range_post_init(self):
        """
        Validate that start_time comes before end_time if both are provided.
        Call this after instantiating the model.
        """
        if self.start_time and self.end_time and self.start_time > self.end_time:
            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`

```python
@field_validator("start_time", "end_time")
@classmethod
def validate_time_range(cls, v: Optional[datetime], info) -> Optional[datetime]:
    """
    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.
    """
    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`

```python
def validate_time_range_post_init(self):
    """
    Validate that start_time comes before end_time if both are provided.
    Call this after instantiating the model.
    """
    if self.start_time and self.end_time and self.start_time > self.end_time:
        raise ValueError("start_time must be before end_time")
```

2.10.3 `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
class AnalyticsResponse(BaseModel):
    """
    Model for analytics data returned in API responses.

    Attributes:
        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.
    """
    id: str
    user_id: str
    device_id: str
    data_type: str
    metrics: Dict[str, Any]
    tags: List[str]
    timestamp: datetime

    model_config = ConfigDict(from_attributes=True)
```

2.10.4 `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`

```python
class AnalyticsUpdate(BaseModel):
    """
    Model for updating analytics information.

    Attributes:
        metrics (Optional[Dict[str, Any]]): Updated metrics.
        tags (Optional[List[str]]): Updated tags for categorizing analytics data.
    """
    metrics: Optional[Dict[str, Any]] = None
    tags: Optional[List[str]] = None

    @field_validator("tags")
    @classmethod
    def validate_tags(cls, v: Optional[List[str]]) -> Optional[List[str]]:
        """
        Validate tags.

        Arguments:
            v (Optional[List[str]]): Tags to be validated.

        Returns:
            Optional[List[str]]: Validated tags.

        Raises:
            ValueError: If tags contain empty strings.
        """
        if v is not None:
            if any(not tag.strip() for tag in v):
                raise ValueError("Tags cannot contain empty strings.")
            return [tag.strip() for tag in v]
        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`

```python
@field_validator("tags")
@classmethod
def validate_tags(cls, v: Optional[List[str]]) -> Optional[List[str]]:
    """
    Validate tags.

    Arguments:
        v (Optional[List[str]]): Tags to be validated.

    Returns:
        Optional[List[str]]: Validated tags.

    Raises:
        ValueError: If tags contain empty strings.
    """
    if v is not None:
        if any(not tag.strip() for tag in v):
            raise ValueError("Tags cannot contain empty strings.")
        return [tag.strip() for tag in v]
    return v
```

2.10.5 `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`

```python
class CreateAnalytics(BaseModel):
    """
    Model for analytics data input.

    Attributes:
        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.
    """
    user_id: str
    device_id: str
    data_type: str
    metrics: Dict[str, Any]
    tags: Optional[List[str]] = []

    @field_validator("user_id", "device_id")
    @classmethod
    def validate_id(cls, v: str) -> str:
        """
        Validate ID fields.

        Arguments:
            v (str): ID to be validated.

        Returns:
            str: Validated ID.

        Raises:
            ValueError: If ID is empty or doesn't match UUID format.
        """
        if not v:
            raise ValueError("ID cannot be empty.")

        # Try to parse as UUID to ensure valid format
        try:
            uuid.UUID(v)
        except ValueError as exc:
            raise ValueError("ID must be a valid UUID format.") from exc

        return v

    @field_validator("data_type")
    @classmethod
    def validate_data_type(cls, v: str) -> str:
        """
        Validate data_type field.

        Arguments:
            v (str): data_type to be validated.

        Returns:
            str: Validated data_type.

        Raises:
            ValueError: If data_type is empty or invalid.
        """
        valid_types = ["energy", "usage", "temperature", "humidity", "motion", "light", "other"]
        if not v:
            raise ValueError("Data type cannot be empty.")

        if v.lower() not in valid_types:
            raise ValueError(f"Data type must be one of: {', '.join(valid_types)}")

        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`

```python
@field_validator("data_type")
@classmethod
def validate_data_type(cls, v: str) -> str:
    """
    Validate data_type field.

    Arguments:
        v (str): data_type to be validated.

    Returns:
        str: Validated data_type.

    Raises:
        ValueError: If data_type is empty or invalid.
    """
    valid_types = ["energy", "usage", "temperature", "humidity", "motion", "light", "other"]
    if not v:
        raise ValueError("Data type cannot be empty.")

    if v.lower() not in valid_types:
        raise ValueError(f"Data type must be one of: {', '.join(valid_types)}")

    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`

```python
@field_validator("user_id", "device_id")
@classmethod
def validate_id(cls, v: str) -> str:
    """
    Validate ID fields.

    Arguments:
        v (str): ID to be validated.

    Returns:
        str: Validated ID.

    Raises:
        ValueError: If ID is empty or doesn't match UUID format.
    """
    if not v:
        raise ValueError("ID cannot be empty.")

    # Try to parse as UUID to ensure valid format
    try:
        uuid.UUID(v)
    except ValueError as exc:
        raise ValueError("ID must be a valid UUID format.") from exc

    return v
```

2.11 `app.models.automation`

Models for automation validation.

2.11.1 `app.models.automation.ActionType`

Bases: `str`, `Enum`

Enum for different types of automation actions.

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

```python
class ActionType(str, Enum):
    """
    Enum for different types of automation actions.
    """
    DEVICE_CONTROL = "device_control"
    NOTIFICATION = "notification"
    SCENE_ACTIVATION = "scene_activation"
    ENERGY_MANAGEMENT = "energy_management"
```

2.11.2 `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
class AutomationDB(BaseModel):
    """
    Internal model representing automation data in the database.

    Attributes:
        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.
    """
    id: str = Field(default_factory=lambda: str(uuid.uuid4()))
    name: str
    description: str
    user_id: str
    device_id: str
    enabled: bool = True
    trigger_type: TriggerType
    trigger_data: Dict[str, Any]
    action_type: ActionType
    action_data: Dict[str, Any]
    conditions: Optional[List[Dict[str, Any]]] = None
    created: datetime = Field(default_factory=datetime.utcnow)
    updated: Optional[datetime] = None
    last_triggered: Optional[datetime] = None
    execution_count: int = 0

    model_config = ConfigDict(from_attributes=True)
```

2.11.3 `app.models.automation.AutomationDetailResponse`

Bases: `AutomationResponse`

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

ditional 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`

```
class AutomationDetailResponse(AutomationResponse):
    """
    Detailed model for automation data returned in API responses.
    Extends AutomationResponse to include configuration details.

    Additional 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.
    """
    trigger_data: Dict[str, Any]
    action_data: Dict[str, Any]
    conditions: Optional[List[Dict[str, Any]]] = None
    updated: Optional[datetime] = None
```

2.11.4 `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`

```python
class AutomationResponse(BaseModel):
    """
    Model for automation data returned in API responses.

    Attributes:
        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.
    """
    id: str
    name: str
    description: str
    user_id: str
    device_id: str
    enabled: bool
    trigger_type: TriggerType
    action_type: ActionType
    created: datetime
    last_triggered: Optional[datetime] = None
    execution_count: int

    model_config = ConfigDict(from_attributes=True)
```

2.11.5 `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`

```python
class AutomationUpdate(BaseModel):
    """
    Model for updating automation information.

    Attributes:
        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.
    """
    name: Optional[str] = None
    description: Optional[str] = None
    enabled: Optional[bool] = None
    trigger_type: Optional[TriggerType] = None
    trigger_data: Optional[Dict[str, Any]] = None
    action_type: Optional[ActionType] = None
    action_data: Optional[Dict[str, Any]] = None
    conditions: Optional[List[Dict[str, Any]]] = None

    @field_validator("name")
    @classmethod
    def validate_name(cls, v: Optional[str]) -> Optional[str]:
        """
        Validate automation name.

        Arguments:
            v (Optional[str]): Name to be validated.

        Returns:
            Optional[str]: Validated name.

        Raises:
            ValueError: Validation encountered a missing requirement.
        """
        if isinstance(v, str):
            if len(v) < 3:
                raise ValueError("Automation name must be at least 3 characters long.")
            if len(v) > 50:
                raise ValueError("Automation name must be less than 50 characters long.")

        return v

    @field_validator("description")
    @classmethod
    def validate_description(cls, v: Optional[str]) -> Optional[str]:
        """
        Validate automation description.

        Arguments:
            v (Optional[str]): Description to be validated.

        Returns:
            Optional[str]: Validated description.

        Raises:
            ValueError: Validation encountered a missing requirement.
        """
        if isinstance(v, str) and len(v) > 500:
            raise ValueError("Automation description must be less than 500 characters long.")

        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`

```python
@field_validator("description")
@classmethod
def validate_description(cls, v: Optional[str]) -> Optional[str]:
    """
    Validate automation description.

    Arguments:
        v (Optional[str]): Description to be validated.

    Returns:
        Optional[str]: Validated description.

    Raises:
        ValueError: Validation encountered a missing requirement.
    """
    if isinstance(v, str) and len(v) > 500:
        raise ValueError("Automation description must be less than 500 characters long.")

    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`

```python
@field_validator("name")
@classmethod
def validate_name(cls, v: Optional[str]) -> Optional[str]:
    """
    Validate automation name.

    Arguments:
        v (Optional[str]): Name to be validated.

    Returns:
        Optional[str]: Validated name.

    Raises:
        ValueError: Validation encountered a missing requirement.
    """
    if isinstance(v, str):
        if len(v) < 3:
            raise ValueError("Automation name must be at least 3 characters long.")
        if len(v) > 50:
            raise ValueError("Automation name must be less than 50 characters long.")

    return v
```

2.11.6 `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.

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

```python
class CreateAutomation(BaseModel):
    """
    Model for automation creation input.

    Attributes:
        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.
    """
    name: str
    description: str
    user_id: str
    device_id: str
    enabled: bool = True
    trigger_type: TriggerType
    trigger_data: Dict[str, Any]
    action_type: ActionType
    action_data: Dict[str, Any]
    conditions: Optional[List[Dict[str, Any]]] = None

    @field_validator("name")
    @classmethod
    def validate_name(cls, v: str) -> str:
        """
        Validate automation name.

        Arguments:
            v (str): Name to be validated.

        Returns:
            str: Validated name.

        Raises:
            ValueError: Validation encountered a missing requirement.
        """
        if len(v) < 3:
            raise ValueError("Automation name must be at least 3 characters long.")
        if len(v) > 50:
            raise ValueError("Automation name must be less than 50 characters long.")

        return v

    @field_validator("description")
    @classmethod
    def validate_description(cls, v: str) -> str:
        """
        Validate automation description.

        Arguments:
            v (str): Description to be validated.

        Returns:
            str: Validated description.

        Raises:
            ValueError: Validation encountered a missing requirement.
        """
        if len(v) > 500:
            raise ValueError("Automation description must be less than 500 characters long.")

        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*

Returns:

Name	Type	Description
`str`	`str`	Validated description.

Raises:

Type	Description
`ValueError`	Validation encountered a missing requirement.

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

```python
@field_validator("description")
@classmethod
def validate_description(cls, v: str) -> str:
    """
    Validate automation description.

    Arguments:
        v (str): Description to be validated.

    Returns:
        str: Validated description.

    Raises:
        ValueError: Validation encountered a missing requirement.
    """
    if len(v) > 500:
        raise ValueError("Automation description must be less than 500 characters long.")

    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`

```python
@field_validator("name")
@classmethod
def validate_name(cls, v: str) -> str:
    """
    Validate automation name.

    Arguments:
        v (str): Name to be validated.

    Returns:
        str: Validated name.

    Raises:
        ValueError: Validation encountered a missing requirement.
    """
    if len(v) < 3:
        raise ValueError("Automation name must be at least 3 characters long.")
    if len(v) > 50:
        raise ValueError("Automation name must be less than 50 characters long.")

    return v
```

2.11.7 `app.models.automation.TriggerType`

Bases: `str`, `Enum`

Enum for different types of automation triggers.

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

```python
class TriggerType(str, Enum):
    """
    Enum for different types of automation triggers.
    """
    TIME = "time"
    SENSOR = "sensor"
    MANUAL = "manual"
    DEVICE_STATE = "device_state"
    LOCATION = "location"
    WEATHER = "weather"
```

2.12 `app.models.device`

Model for device validation & storage.

2.12.1 `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
class CreateDevice(BaseModel):
    """
    Model for device registration input.

    Attributes:
        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.
    """
    name: str
    type: DeviceType
    user_id: str
    room_id: Optional[str] = None
    ip_address: Optional[str] = None
    mac_address: Optional[str] = None
    manufacturer: Optional[str] = None
    model: Optional[str] = None
    firmware_version: Optional[str] = None
    settings: Optional[Dict[str, Any]] = None

    @field_validator("name")
    @classmethod
    def validate_name(cls, n: str) -> str:
        """
        Validate device name according to requirements.

        Args:
            name (str): Device name to be validated.

        Returns:
            str: Validated device name.

        Raises:
            ValueError: Validation encountered a missing requirement.
        """
        if len(n) < 1:
            raise ValueError("Device name can't be empty.")
        if len(n) > 100:
            raise ValueError("Device name must be less than 100 characters long.")

        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`

```python
@field_validator("name")
@classmethod
def validate_name(cls, n: str) -> str:
    """
    Validate device name according to requirements.

    Args:
        name (str): Device name to be validated.

    Returns:
        str: Validated device name.

    Raises:
        ValueError: Validation encountered a missing requirement.
    """
    if len(n) < 1:
        raise ValueError("Device name can't be empty.")
    if len(n) > 100:
        raise ValueError("Device name must be less than 100 characters long.")

    return n
```

2.12.2 `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`

```python
class DeviceDB(BaseModel):
    """
    Internal model representing device data in the database

    Attributes:
        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.
    """
    id: str
    name: str
    type: DeviceType
    user_id: str
    room_id: Optional[str] = None
    ip_address: Optional[str] = None
    mac_address: Optional[str] = None
    manufacturer: Optional[str] = None
    model: Optional[str] = None
    firmware_version: Optional[str] = None
    settings: Optional[Dict[str, Any]] = None
    status: DeviceStatus = DeviceStatus.OFFLINE
    last_online: Optional[datetime] = None
    created: datetime = Field(default_factory=datetime.utcnow)
    updated: Optional[datetime] = None
    capabilities: List[str] = []

    model_config = ConfigDict(from_attributes=True)
```

2.12.3 `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
class DeviceResponse(BaseModel):
    """
    Model for device data returned in API responses.

    Attributes:
        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.
    """
    id: str
    name: str
    type: DeviceType
    user_id: str
    room_id: Optional[str] = None
    manufacturer: Optional[str] = None
    model: Optional[str] = None
    status: DeviceStatus
    last_online: Optional[datetime] = None
    created: datetime
    capabilities: List[str] = []
```

2.12.4 `app.models.device.DeviceStatus`

Bases: `str`, `Enum`

Enumeration of possible device statuses.

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

```python
class DeviceStatus(str, Enum):
    """
    Enumeration of possible device statuses.
    """
    ONLINE = "online"
    OFFLINE = "offline"
    ERROR = "error"
    MAINTENANCE = "maintenance"
```

2.12.5 `app.models.device.DeviceType`

Bases: `str`, `Enum`

Enumeration of supported device types.

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

```python
class DeviceType(str, Enum):
    """
    Enumeration of supported device types.
    """
    LIGHT = "light"
    THERMOSTAT = "thermostat"
    LOCK = "lock"
    CAMERA = "camera"
    SENSOR = "sensor"
    SWITCH = "switch"
    OUTLET = "outlet"
    SPEAKER = "speaker"
    OTHER = "other"
```

2.12.6 `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`

```python
class DeviceUpdate(BaseModel):
    """
    Model for updating device information.

    Attributes:
        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.
    """
    name: Optional[str] = None
    room_id: Optional[str] = None
    ip_address: Optional[str] = None
    firmware_version: Optional[str] = None
    settings: Optional[Dict[str, Any]] = None
    status: Optional[DeviceStatus] = None

    @field_validator("name")
    @classmethod
    def validate_name(cls, n: Optional[str]) -> Optional[str]:
        """
        Validate device name according to requirements.

        Args:
            n (Optional[str]): Device name to be validated.

        Returns:
            Optional[str]: Validated device name.

        Raises:
            ValueError: Validation encountered a missing requirement.
        """
        if n is None:
            return None

        if len(n) < 1:
            raise ValueError("Device name can't be empty.")
        if len(n) > 100:
            raise ValueError("Device name must be less than 100 characters long.")

        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`

```
@field_validator("name")
@classmethod
def validate_name(cls, n: Optional[str]) -> Optional[str]:
    """
    Validate device name according to requirements.

    Args:
        n (Optional[str]): Device name to be validated.

    Returns:
        Optional[str]: Validated device name.

    Raises:
        ValueError: Validation encountered a missing requirement.
    """
    if n is None:
        return None

    if len(n) < 1:
        raise ValueError("Device name can't be empty.")
    if len(n) > 100:
        raise ValueError("Device name must be less than 100 characters long.")

    return n
```

2.13 `app.models.goal`

Models for energy goal validation and storage.

2.13.1 `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`